**VŠB - Technická univerzita Ostrava**
**Fakulta elektrotechniky a informatiky**
Katedra informatiky

# Microsoft Robotics Developer Studio - jazyk VPL
# Microsoft Robotics Developer Studio - VPL

2011 Bc. Miroslav Novotný

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Miroslav Novotný**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Microsoft Robotics Developer Studio - jazyk VPL
Microsoft Robotics Developer Studio - VPL

Zásady pro vypracování:

Microsoft Robotics Developer Studio osahuje nástroj pojmenovaný Visual Programming Language. Tento nástroj, sloužící ke grafickému programování, je založen na propracovaném frameworku. Do VPL lze snadno přidávat vlastní komponenty (služby), napsané např. v C# a ty lze pospojovat do složitějších programů. Naopak lze diagramy konvertovat na služby. Cílem práce je popsat VPL a navrhnout a naimplementovat ukázkové aplikace.
1. Seznamte se s Microsoft Robotics Developer Studiem.
2. Vhodným způsobem popište VPL a framework, sloužící k práci s VPL.
3. Navrhněte a naimplementujte vhodné úlohy demonstrující možnosti jazka VPL.
4. Tyto úlohy a použití VPL pro jejich realizaci přehledně zdokumentujte.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **RNDr. Eliška Ochodková, Ph.D.**

Datum zadání: 20.11.2009
Datum odevzdání: 06.05.2011


doc. Dr. Ing. Eduard Sojka
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne 5. května 2011 .....

# Abstrakt

Microsoft Robotics Developer Studio je volně dostupné programovací prostředí, založené na architektuře .NET. Jak už z názvu je patrné, MRDS je určené k vývoji aplikací zaměřených především na robotiku. Jeho možnosti jsou natolik rozsáhlé, že se s jeho pomocí stává tento obor lehce dostupný nejen pro profesionály, ale i pro pouhé nadšence.



**Klíčová slova**: Microsoft Robotics Developer Studio, Concurrency and Coordination Runtime, Decentralized Software Services, Visual Programming Language, Visual Simulation Environment.



# Abstract

Microsoft Robotics Developer Studio is a freely available programming environment based on the .NET architecture. As the name suggests, MRDS is intended to develop applications focused mainly on robotics. The possibilities are so extensive, that with the help of this branch becomes easily available not only for professionals, but also for mere enthusiasts.



**Keywords**: Microsoft Robotics Developer Studio, Concurrency and Coordination Runtime, Decentralized Software Services, Visual Programming Language, Visual Simulation Environment.

# Seznam použitých symbolů a zkratek

| | |
|---|---|
| MRDS | - Microsoft Robotics Developer Studio |
| CCR | - Concurrency and Coordination Runtime |
| DSS | - Decentralized Software Services |
| VPL | - Visual Programming Language |
| VSE | - Visual Simulation Environment |
| XML | - Extensible Markup Language |
| DSSME | - Decentralized Software Services Manifest Editoru |

# Obsah

# 1 Úvod

Microsoft Robotics Developer Studio je integrované vývojové prostředí, postavené na architektuře .NET. S jeho pomocí lze navrhovat, spouštět a ladit aplikace, které vyžadují souběžné vykonávání na sobě zdánlivě nezávislých úloh. Vše je zaměřeno především na oblast robotiky, kde bezchybné řešení problémů se souběžně běžícími úlohami, je nezbytným základem úspěchu. Kromě nejrůznějších nástrojů a běhového prostředí, přichází MRDS s mnoha příklady a návody, které umožňují psát aplikace jak amatérským nadšencům, tak i profesionálům.

Jednou ze zajímavých součástí MRDS, je Microsoft Visual Programming Language, který si přiblížíme při řešení problému přístupu více vláken ke sdílenému zdroji, známým pod názvem „problém večeřících filozofů“. K tomu, aby bylo možné VPL správně použít, je nutné se nejprve podrobně seznámit s Concurrency and Coordination Runtime a Decentralized Software Services, neboť s těmito technologiemi Visual Programming Language úzce spolupracuje.

Nejprve si ve druhé kapitole krátce popíšeme nástroje a služby, které budeme mít k dispozici. Ve třetí kapitole se podrobněji podíváme na Concurrency and Coordination Runtime, který slouží jako základ celého Microsoft Robotics Developer Studia. Ve čtvrté kapitole se seznámíme s Decentralized Software Services, které jsou nadstavbou CCR, a teprve v páté kapitole se dostaneme k Visual Programming Language, neboť bez pochopení CCR a DSS by nebylo možné problém večeřících filozofů správně vyřešit. Samotné řešení výše uvedeného problému, tedy uvedeme v kapitole šesté. V kapitole sedné, popíšeme instalaci MRDS. Zhodnocení celého našeho snažení, shrneme v osmé kapitole se závěrem. Kapitola devátá bude uvádět všechny zdroje, ze kterých se čerpalo. Výpis obsahu přiloženého CD, bude uveden v kapitole Příloha A.

# 2 Jaké nástroje a služby máme k dispozici?

Microsoft Robotics Developer Studio je vývojové a běhové prostředí, určené k realizaci aplikací zaměřených především na robotiku. Nyní si tedy stručně představme jeho hlavní nástroje a služby. Podrobnější informace jsou k dispozici na stránkách Microsoft Robotics Developer Studio Online Documentation [2].

## 2.1 Concurrency and Coordination Runtime

Concurrency and Coordination Runtime, označovaný zkratkou CCR, nám poskytuje programovací model, založený na předávání zpráv. Tento model umožňuje předávání zpráv, bez nutnosti se neustále zabývat vlákny, zámky, či semafory. Můžeme tak relativně jednoduše vyvíjet aplikace, běžící ve více vláknech, nebo aplikace orientující se na služby, neboť CCR nám usnadňuje správu asynchronních a souběžně vykonávaných operací. Lze tak paralelně využívat skoro jakýkoliv hardware.

## 2.2 Decentralized Software Services

Pod zkratkou DSS se skrývají decentralizované programové služby, které lze chápat jako jakési zjednodušené řešení distribuovaných systémů, které své uplatnění nacházejí především ve světě webově zaměřených službách. DSS své rozhraní vystavují pomocí XML dokumentů, které lze zpracovat přímo programově, nebo k nim lze přistupovat pomocí webového prohlížeče. Tím je umožněno psát aplikace, běžící nejenom na jednom počítači, ale i na více mezi sebou propojenými servery. Hlavní využití DSS se předpokládá pro vzájemnou komunikaci mezi jednotlivými částmi robotů.

## 2.3 DSS Manifest Editor

Úkolem DSS Manifest Editoru je umožnit pomocí grafického uživatelského rozhraní, konfiguraci, nasazení a správu běžících DSS služeb. Pomocí DSSME lze sestavit jak aplikace a služby napsané ve Visual Studiu, tak pomocí VPL. Instalace a konfigurace služeb je zajištěna tak, že pomocí DSSME se určí, z jakých služeb má být aplikace sestavena, kde se jednotlivé služby nacházejí a jaký je jejich počáteční stav. Takto lze realizovat vysoce škálovatelná řešení, která zasahují do nejrůznějších odvětví, kterými mohou být například vědecké výpočty, průmyslová automatizace, finanční analýzy a mnoha dalšími, kde jsou kladeny vysoké požadavky na distribuované a paralelně běžící systémy.

## 2.4 Visual Programming Language

Microsoft Visual Programming Language, je grafické vývojové prostředí, pro realizaci CCR/DSS programů. Na rozdíl od konvenčního pojetí psaní programů, je výsledný systém realizován pomocí jednotlivých komponent řazených do větších celků, na základě logického toku dat. Komponenty poskytující různé služby, se vybírají z palety komponent a pomocí myši se seskupují do požadovaného toku dat. Díky schopnosti VPL vygenerovat z navrženého diagramu kód v jazyce C#, lze realizaci výsledného CCR/DSS programu, celkem znatelně urychlit. Předpokladem je však existence potřebných komponent, které si v některých případech, musíme sami naimplementovat.

## 2.5 Visual Simulation Environment

Pomocí VSE lze realizovat simulaci chování robotů, prostředí, či jiných objektů, využívajících trojrozměrné zobrazení. Zobrazení scény je realizováno pomocí Microsoft XNA Frameworku a o fyzikální chování objektů v zobrazené scéně, se stará NVIDIA PhysX™ Technologies. Díky této technologii, se mohou zobrazované objekty vzájemně mezi sebou nebo okolním prostředím, ovlivňovat způsobem, který se velmi přibližuje chování v reálném světě. Mimo samotného zobrazení scény, Visual Simulation Environment umožňuje například i záznam, přehrávání a samozřejmě i vytvoření nějaké konkrétní scény.

# 3 Concurrency and Coordination Runtime

Concurrency and Coordination Runtime je základem celého Microsoft Robotics Developer Studia. Je tedy nezbytně nutné, pochopit jeho hlavní principy. Pokud budete potřebovat více informací, než je zde uvedeno, tak je lze získát přímo se stránek, z nichž tato kapitola čerpá [3].

## 3.1 Úvod do CCR

Nezbytnou součástí programů, bývá správa asynchronních operací, řešení souběžnosti, paralelní využití hardware a ošetření chybových stavů. Na správném návrhu řešení těchto problémových oblastí, závisí celkové chování výsledného programu. A právě proto, že je nutné se těmito oblastmi neustále zabývat, tak nám CCR nabízí způsob, jak se s nimi vypořádat. Tím se sjednotí postup řešení ve všech programech, kde se CCR použije, ale hlavně je možné se pak soustředit především na problematiku, kterou má program řešit, a nemusíme se neustále zabývat problémem správné synchronizace vláken, a podobně.

Konkrétně je Concurrency and Coordination Runtime knihovna dll, ke které lze přistupovat z jakéhokoliv programovacího jazyka, po jehož přeložení vzniká program využívající .NET Common Language Runtime (CLR). Těmito jazyky jsou například Visual Basic .NET, J#, F# nebo C#.

### 3.1.1 Problémové oblasti

Asynchronie – S asynchronním chováním programů, se nejčastěji setkáváme u uživatelského rozhranní, kde díky tomu uživatel nemusí čekat na dokončení vstupně výstupních operací, nebo při vzájemné komunikaci programů napříč sítí. Tento komfort je však vykoupen sníženou čitelností kódu, poněvadž logika programu je rozdělena na metody vytvářející asynchronní volání a zpětné volání metod. Ošetření chyb, které během asynchronního volání mohou nastat, rovněž vede k nárůstu kódu a celkové složitosti.

Souběh – Aby program efektivně využíval přidělené zdroje, tak často bývá rozdělen do několika na sobě zdánlivě nezávislých a paralelně vykonávaných částí. Tyto jednotlivé části však potřebují mezi sebou komunikovat a nezřídka také pracují se stejnými zdroji, jako je například operační paměť, datové úložiště, nebo připojené periférie. Je tedy nutné zajistit, aby se hodnota proměnných během vykonávání kódu nečekaně neměnila, nebo se současně neposílala data na nějaké zařízení, protože by to mohlo vést k pomíchání dat, nebo dokonce k poničení hardware.

Ošetření chybových stavů – Během paralelního vykonávání jakékoliv části programu, může dojít k chybovému stavu. Proto je velmi důležité mít k dispozici mechanizmus, který dokáže na vzniklou situaci reagovat, umožnit zotavení chybující části, nebo přizpůsobit vzniklé situaci běh ostatních částí programu.

### 3.1.2 Kdy použít CCR?

Pokud je možné logiku programu rozdělit na části, které běží paralelně a mohou mezi sebou komunikovat pouze pomocí zpráv, tak je velmi pravděpodobné, že bude možné použít služeb knihovny CCR. Pro lepší představu, jak toho dosáhnout, si v následujících kapitolách Concurrency and Coordination Runtime představíme konkrétněji.

## 3.2 Představení CCR API

Concurrency and Coordination Runtime má na starosti následující funkce:

Pomocí třídy Port a PortSet, se zajišťuje řazení příchozích prvků do kolekce typu FIFO. Prvkem zde chápeme takový datový typ, který umí požadovaný Port přijmout. Pro zlepšení výkonu, lze použít filtrování příchozích zpráv pomocí tříd nazývaných Arbiters.

Uživatelský kód se vykonává pomocí tříd, které se souhrnně nazývají Arbiters. Tyto třídy pomocí delegátů volají metody, které vykonávají konkrétní uživatelský kód. Od Arbiters lze vytvářet odvozené třídy, které pak mohou překrýt vybrané metody a implementovat tak vlastní logiku.

CCR také zajišťuje vyrovnávání zátěže, a to pomocí tříd Dispatcher, DispatcherQueue a Task. Tím je samotná logika programu odstíněna od nutnosti zabývat se optimálním zatížením systému. O tom, kdy se vykoná nějaká konkrétní úloha (Task), rozhoduje třída Dispatcher. Třída Dispatcher má při spuštění nadefinováno, kolika úlohám má zajišťovat běh. Jediným způsobem, jak lze ovlivnit chování objektu třídy Dispatcher, je objekt třídy DispatcherQueue, pomocí kterého můžeme přizpůsobit vykonávání úloh tak, aby byla zátěž systému optimální.

## 3.3 Třída Port a PortSet

Port je třídou, která se používá pro komunikaci mezi libovolnými dvěma komponentami. Implementuje rozhranní IPortReceive a IPort, která definují metody například pro přidání prvku, nebo jeho odebrání. K dispozici máme třídu Port, která pracuje pouze s jedním generickým typem prvku. Pokud nám to nevyhovuje, tak si můžeme vytvořit vlastní třídu, implementující rozhranní IPortReceive a IPort.

### 3.3.1 Odesílání prvků na port

Přidání prvku do portu, probíhá asynchronně. Pokud není k portu přiřazen žádný objekt pro filtrování (Arbiter), tak je prvek vložen do fronty. Pokud jsou nějaké objekty Arbiter přiřazené, tak je jim postupně přijatý prvek předáván, a na základě jeho vyhodnocení se rozhodne, zda má být vytvořena nějaká úloha (Task) a naplánováno její spuštění. Toto vyhodnocení je rychlá neblokující operace, která je však zdrojem asynchronního chování. Programy tedy odesílají prvky na porty, kde se po jejich vyhodnocení vytvářejí úlohy, které mohou být vykonávané v jiném vlákně, než kterým byl prvek přijat.

Na jednoduchém příkladu, který je uveden ve výpisu 1, si ukážeme, jak se na port odešle prvek typu integer s hodnotou 4, a na konzolu se zobrazí počet prvků, které port obsahuje ve své frontě:

```
// Vytvoření portu, který přijímá instance třídy System.Int32.
Port<int> integerPort = new Port<int>();

// Odeslání čísla 4 na port.
integerPort.Post(4);

// Zobrazení počtu prvků, které obsahuje port ve své frontě.
Console.WriteLine(integerPort.ItemCount);
```

Výpis 1: Odeslání prvku na port

### 3.3.2 Načtení prvků z portu

Máme k dispozici dva způsoby, jak můžeme získat prvky z fronty portu:

Pokud je port použit jako pasivní fronta, to znamená, že nejsou vytvářeny žádné úlohy po vložení prvků, tak lze k načtení prvků z portu použít metodu Test. Pokud port obsahuje nějaký prvek, tak je první prvek ve frontě předán jako výstupní parametr, a návratová hodnota metody Test, bude true. Pasivní frontu lze použít v těch případech, kdy chceme port použít pouze ke sběru prvků, a máme nějaký vlastní kód, který čte a zpracovává jednotlivé prvky portu. Po každém načtení prvku, je tento prvek z portu odebrán.

Ve výpisu 2 je předvedeno odeslání prvku na port a jeho načtení z portu, pomocí metody Test. Také si předvedeme alternativní způsob načtení prvku z portu, pomocí použití operátoru přiřazení:

```
// Vytvoření portu, který přijímá instance třídy System.Int32.
Port<int> integerPort = new Port<int>();

// Odeslání čísla 4 na port.
integerPort.Post(4);

// Zobrazení počtu prvků, které obsahuje port ve své frontě.
Console.WriteLine(integerPort.ItemCount);

// Načtení prvku pomocí metody Test.
int item;
bool hasItem = integerPort.Test(out item);
if (hasItem)
{
    Console.WriteLine("Načtený prvek z portu: " + item);
}

integerPort.Post(7);
// Místo metody Test, lze použít operátor přiřazení.
var nextItem = integerPort;

Console.WriteLine("Načtený prvek z portu: " + nextItem);
```

Výpis 2: Načtení prvku z portu

Častější použití portu je jako aktivní fronta, kdy k portu je zaregistrován jeden, nebo více arbitrů, kteří na základě přijatých prvků vytvářejí úlohy a plánují jejich spuštění. Tyto úlohy pak mohou běžet i paralelně.

Následující příklad ve výpisu 3 nepoužívá k načtení prvku z portu metodu Test, ale metodu Arbiter.Activate, která k portu pomocí delegáta zaregistruje přijímač portu, kterým v našem případě bude anonymní metoda. Tělo anonymní metody bude vykonáno v samostatném vlákně, které je spravováno pomocí instance třídy DispatcherQueue. Tuto třídu si důkladněji představíme později. Nyní je hlavní si uvědomit, že delegovaná metoda bude probíhat paralelně s hlavní metodou programu. Poté, co se delegovaná metoda spustí, tak bude přijímač automaticky z portu odebrán.

```
// Vytvoření portu, který přijímá instance třídy System.Int32.
Port<int> integerPort = new Port<int>();

// Odeslání čísla 4 na port.
integerPort.Post(4);

// Zobrazení počtu prvků, které obsahuje port ve své frontě.
Console.WriteLine(integerPort.ItemCount);

// Vytvoření instance třídy Dispatcher a DispatcherQueue,
// pro plánování a vykonávání úloh.
Dispatcher dispatcher = new Dispatcher();
DispatcherQueue dispatcherQueue = new DispatcherQueue("DispatcherQueue",
dispatcher);

// Načtení prvku pomocí zaregistrovaného přijímače portu.
Arbiter.Activate(
    dispatcherQueue,
    integerPort.Receive(delegate (int item) // Anonymní metoda.
    {
        Console.WriteLine("Načtený prvek: " + item);
    }
));
// Následující kód by běžel paralelně s kódem v anonymní metodě.
```

Výpis 3: Načtení prvku z portu anonymní metodou

### 3.3.3 Sledování stavu portu

Pomocí metody ToString třídy Port, můžeme jednoduše zobrazit informaci o počtu prvků ve frontě, a hierarchii přijímačů zaregistrovaných k portu.

Metoda ItemCount třídy Port, nám zobrazuje počet prvků ve frontě. Je ale důležité mít na paměti, že pokud je k portu pomocí arbitra zaregistrován minimálně jeden přijímač portu, tak bude metoda ItemCount, vždy vracet nulu. V tomto případě se totiž prvky po spuštění kódu přijímačů, z fronty automaticky odebírají.

### 3.3.4 Třída PortSet

Vzhledem k tomu, že třída Port umí pracovat pouze s jedním prvkem, tak je často potřeba vytvářet více portů, a to třeba i pro různé typy prvků. Pomocí třídy PortSet lze tyto porty seskupit do logického celku, a předávat jej jako jeden objekt. CCR komponenty tak místo veřejných metod, mají své rozhranní definované pomocí objektů PortSet s libovolným počtem na sobě nezávislých portů. Těmto portům jsou zasílány zprávy a ty jsou zpracovávány nezávisle na jejich zdroji.

### 3.3.5 Použití třídy PortSet

Ukážeme si dva způsoby, jak vytvořit instanci třídy PortSet. Pokud potřebný počet portů a typy přijímaných prvků, známe již v době kompilace, tak stačí zadat typy do konstruktoru, jak je uvedeno ve výpisu 4:

```
// Vytvoření instance PortSet, pokud předem známe počet potřebných portů.
PortSet portSet = new PortSet<string, int, double>();
portSet.Post("abc");
portSet.Post(4);
portSet.Post(2.71828d);
```

Výpis 4: Vytvoření instance třídy PortSet s předem známými typy prvků

Občas se však může vyskytnout situace, že potřebný počet portů a typy jimi přijímaných prvků, je znám až za běhu aplikace. V takovém případě lze použít konstruktor třídy PortSet, kterému se jako parametr předá pole typů.

Při použití tohoto způsobu vytváření instance z třídy PortSet, je však nutné mít na paměti, že k odesílání prvků na porty nelze použít metodu Post, ale místo ní se musí použít metoda PostUnknownType, nebo TryPostUnknownType, jak ilustruje výpis 5:

```
// Vytvoření instance z PortSet, pomocí pole typů.
PortSet portSet = new PortSet(
    typeof(string),
    typeof(int),
    typeof(double)
    );
portSet.PostUnknownType("abc");
portSet.PostUnknownType(4);
portSet.PostUnknownType(2.71828d);
```

Výpis 5: Vytvoření instance z PortSet, pomocí pole typů

Třída PortSet nabízí standardní metody, pomocích kterých lze projít všechny porty. Pokud však potřebujeme pouze zaregistrovat přijímače nějakého konkrétního portu, tak k tomu můžeme použít způsob, který je demonstrován ve výpisu 6:

```
PortSet portSet = new PortSet<int, string, double>();

// Vytvoření instance třídy Dispatcher a DispatcherQueue,
// pro plánování a vykonávání úloh.
Dispatcher dispatcher = new Dispatcher();
DispatcherQueue dispatcherQueue = new DispatcherQueue("DispatcherQueue",
dispatcher);

// Přiřazení přijímače k portu, přijímajícího prvky typu int.
Arbiter.Activate(dispatcherQueue,
    Arbiter.Receive<int>(true, portSet, item => Console.WriteLine(item))
);
```

Výpis 6: Přiřazení přijímače k portu

Jak je vidět, tak k zaregistrování přijímače portu se opět použije metoda Activate třídy Arbiter, avšak k upřesnění portu, ke kterému se má přijímač zaregistrovat, slouží generická metoda Receive třídy Arbiter. Ta v tomto příkladě vrátí port, který přijímá prvky typu int, a k tomuto portu zaregistruje anonymní metodu, která vypisuje hodnotu přijatého prvku.

## 3.4 Synchronizace pomocí CCR

U asynchronního programování je běžné, že program vyvolá nějakou činnost, a pokračuje dále ještě před dokončením této činnosti. Je tak pravděpodobné, že se v jeden okamžik provádí několik činností současně. A právě tady se naráží na problém, že je potřeba řešit vyvolání těchto činností, mít k dispozici způsob, jak ošetřit selhání vzniklé při vykonávání vyvolané činnosti, a v neposlední řadě je nutné zajistit reakci programu na dokončení činnosti tak, aby nedocházelo k problémům se souběžností, kdy se program může pokoušet vykonávat paralelně kód, který pracuje s nějakým sdíleným prostředkem.

CCR nám nabízí svou pomoc především v těchto dvou scénářích:

1. Při koordinaci příchozích požadavků na komponenty, jejichž metody mají být dlouhodobě přístupné. Typickým zástupcem tohoto konceptu jsou například webové služby, které naslouchají na nějakém síťovém portu, a pomocí HTTP se přenášejí data mezi nějakou konkrétní metodou služby a klientem volajícím tuto metodu. Zde tedy můžeme použít CCR port pro přijímání požadavků, které se mohou dále zpracovávat. Při tomto zpracování můžeme využít možnost, že nějaký předem určený ovladač události nebude nikdy spuštěn, pokud se zrovna vykonává nějaký jiný ovladač události.
2. Při koordinaci odpovědí z jednoho nebo více odeslaných požadavků, kdy může být použit více jak jeden typ návratové hodnoty. Typicky se toho dá využít tak, že je jiný typ hodnoty vrácen při úspěšném vykonání požadavku, ale při jeho selhání, se předá jiný typ návratové hodnoty. Zde nám CCR k řešení této situace, nabízí například třídu PortSet.

### 3.4.1 Statická třída Arbiter

Pomocí metod statické třídy Arbiter, se vytvářejí určité typy objekty, které CCR nabízí. Popíšeme si několik základních metod této třídy. Samotné objekty, které se pomocí těchto metod vytvářejí, lze vytvořit i pomocí konstruktorů konkrétních tříd. Pokud ale nevyžadujeme nějaké specifické nastavení vlastností výsledného objektu, tak se v převážné míře dá pro lepší přehlednost kódu, použít následující metody:

Arbiter.FromTask vytvoří instanci třídy Task.
Arbiter.Choice vytvoří instanci třídy Choice.
Arbiter.Receive vytvoří instanci třídy Receiver.
Arbiter.Interleave vytvoří instanci třídy Interleave.
Arbiter.JoinedReceive vytvoří instanci třídy JoinReceiver.
Arbiter.MultipleItemReceive vytvoří instanci třídy JoinSinglePortReceiver.

Další alternativou, jak zlepšit čitelnost kódu, je použití rozšiřujících metod, jejichž podpora byla do jazyka C#, přidána ve verzi 3.0. Jednoduše řečeno, jedná se o způsob, jak lze do již existujících tříd, jejichž ani nemusíme být autory, a tedy třeba ani nemáme jejich zdrojové kódy, přidat vlastní metody. I tento přístup vytváření objektů, si v příkladech předvedeme.

Nejprve si ukážeme, jak posílat na porty požadavky, po jejichž zpracování se volající straně zašle odpověď, která může být dále zpracována. Potom si blíže představíme jednotlivé třídy CCR a ukážeme si, jak lze jejich instance použít.

### 3.4.2 Použití portů pro přijímání požadavků a odesílání odpovědí

CCR je koncipován tak, aby jednotlivé komponenty přijímaly zprávy do fronty, odkud se dále předávají ovladačům událostí k dalšímu zpracování, jak ilustruje výpis 7:

```
/// <summary>
/// Bázová třída pro všechny příchozí zprávy.
/// Definuje port pro odesílání odpovědí.
/// </summary>
public class ServiceOperation
{
    public PortSet<string, Exception> ResponsePort = new PortSet<string,
Exception>();
}

public class Stop : ServiceOperation {}

public class UpdateState : ServiceOperation
{
    public string State;
}

public class GetState : ServiceOperation
{
}

/// <summary>
/// PortSet, který definuje jaké zprávy může služba přijímat.
/// </summary>
public class ServicePort : PortSet<Stop, UpdateState, GetState>
{
}

/// <summary>
/// CCR komponenta, která používá PortSet k předávání zpráv.
/// </summary>
public class SimpleService
{
    ServicePort servicePort;
    DispatcherQueue dispatcherQueue;
    string state;

    public static ServicePort Create(DispatcherQueue dispatcherQueue)
    {
        SimpleService service = new SimpleService(dispatcherQueue);
        service.Initialize();
        return service.servicePort;
    }
```

```
    private void Initialize()
    {
        // Použije se dispatcherQueue pro plánování aktivovaných dvou
        // přijímačů, které budou spuštěny souběžně,
        // a každý z nich bude přijímat jeden typ zpráv.
        Arbiter.Activate(dispatcherQueue,
            Arbiter.Receive<UpdateState>(true,servicePort,UpdateHandler),
            Arbiter.Receive<GetState>(true, servicePort, GetStateHandler)
        );
    }

    private SimpleService(DispatcherQueue dispatcherQueue)
    {
        // Vytvoření instance třídy PortSet, která bude přijímat zprávy.
        this.servicePort = new ServicePort();
        // Uložení odkazu na frontu použité k plánování úloh.
        this.dispatcherQueue = dispatcherQueue;
    }

    void GetStateHandler(GetState getState)
    {
        if (this.state == null)
        {
            // Pokud nebyl stav ještě nastaven,
            // tak bude jako odpověď odeslána výjimka.
            getState.ResponsePort.Post(new InvalidOperationException());
            return;
        }

        // Jako odpověď se vrátí hodnota proměnné state.
        getState.ResponsePort.Post(this.state);
    }

    void UpdateHandler(UpdateState update)
    {
        // Na základě přijaté zprávy, se aktualizuje proměnná state.
        this.state = update.State;

        // Jako odpověď se vrátí hodnota proměnné state.
        update.ResponsePort.Post(this.state);
    }
}
```

Výpis 7: Použití portů pro přijímání požadavků a odesílání odpovědí

Na příkladu jsme si ukázali jak:

- Definovat typy zpráv použitých ke komunikaci mezi komponentami.
- Vytvořit potomka třídy PortSet, který přijímá předem definované typy zpráv.
- Použít statickou metodu Create(), která inicializuje komponentu a vrátí port určený ke komunikaci.
- V privátní metodě Initialize(), zaregistrovat metody, reagující na příchozí zprávy.

### 3.4.3 Třída Receiver

Třída Receiver, jejíž instanci lze vytvořit i pomocí metody Receive třídy Arbiter, má za úkol propojit instanci třídy Port<T> s metodou, která se má zavolat, při příchodu nějakého prvku na port.

Pokud se jako hodnota parametru persist, použije true, tak se delegovaná metoda zavolá při každém příchodu nějakého prvku na port. Při volbě false, se metoda vykoná pouze pro první příchozí prvek, a pak se delegovaná metoda od portu automaticky odregistruje.

Ve výpisu 8 u portu zaregistruje metodu, která bude pomocí konsole zobrazovat prvky přicházející na port. Protože parametr persist je nastaven na true, tak nedojde k automatickému odregistrování metody, po prvním zpracovaném prvku.

```
// Vytvoření portu, který přijímá instance třídy System.Int32.
Port<int> integerPort = new Port<int>();

// Vytvoření instance třídy Dispatcher a DispatcherQueue,
// pro plánování a vykonávání úloh.
Dispatcher dispatcher = new Dispatcher();
DispatcherQueue dispatcherQueue = new DispatcherQueue("DispatcherQueue",
dispatcher);

// Registrace delegované metody.
bool persist = true;
Arbiter.Activate(
   dispatcherQueue,
   Arbiter.Receive(
       persist,
       integerPort,
       item => Console.WriteLine(item)
   )
);

// Odeslání čísla 4 na port.
integerPort.Post(4);
```

Výpis 8: Použití třídy Receiver, pomocí metody Arbiter.Receive()

Ukažme si, jak by se vytvořila a použila instance třídy Receiver, pomocí konstruktoru, tedy bez volání metody Arbiter.Receive(), jak ilustruje výpis 9.

```
// Vytvoření instance třídy Receiver, pomocí konstruktoru.
Receiver persistedReceiver = new Receiver<int>(
       true, // Definovaná úloha se vykoná pro všechny příchozí prvky.
       integerPort,
       null,
       new Task<int>(item => Console.WriteLine(item)) // Definování úlohy.
    );

Arbiter.Activate(dispatcherQueue, persistedReceiver);
```

Výpis 9: Použití třídy Receiver, bez použití metody Arbiter.Receive()

### 3.4.4 Třída Choice

Instance třídy Choice jsou objekty, do kterých mohou být vnořené jiné objekty, jako například instance třídy Receiver nebo Join. Lze tak vytvářet požadovanou hierarchii a zvolit podle potřeby požadovanou větev objektů. Tak se budou volat pouze metody, zaregistrované pomocí objektů ve vybrané větvi.

Ve výpisu 10 si ukážeme jeden ze způsobů, jak lze třídu Choice použít. Pomocí rozšiřující metody třídy PortSet, vytvoříme instanci třídy Choice, která bude obsahovat dvě větvě. V jedné větvi se pomocí konsole zobrazí hodnota prvku přijatého portem, v druhé větvi se zobrazí zpráva výjimky, která bude portem přijatá. Přestože jsou zde použité pouze dvě větve, tak ve skutečnosti třída Choice umožňuje vytvořit jejich libovolný počet.

```
// Vytvoření instance třídy Dispatcher a DispatcherQueue,
// pro plánování a vykonávání úloh.
Dispatcher dispatcher = new Dispatcher();
DispatcherQueue dispatcherQueue = new DispatcherQueue("DispatcherQueue",
dispatcher);

// Vytvoření služby naslouchající na portu.
ServicePort servicePort = SimpleService.Create(dispatcherQueue);

// Vytvoření požadavku.
GetState getState = new GetState();

// Odeslání požadavku.
servicePort.Post(getState);

// Pomocí rozšiřující metody třídy PortSet, se vytvoří instance
// třídy Choice obsahující dvě větve. O tom, která větev bude použita,
// se rozhodne na základě typu prvku přijatého portem.
Arbiter.Activate(
    dispatcherQueue,
    getState.ResponsePort.Choice(
        s => Console.WriteLine(s), // Větev pro zobrazení hodnot.
        ex => Console.WriteLine(ex) // Větev pro zobrazení výjimky.
));
```

Výpis 10: Použití třídy Choice

Alternativou k rozšiřující metodě třídy PortSet, by bylo vytvoření dvou objektů typu Receiver. Jeden objekt by registroval metodu pro zobrazení hodnot přijatého prvku, druhý objekt by registroval metodu pro zobrazení zprávy z výjimky, a oba objekty by se jako pole předaly jako parametr konstruktoru třídy Choice.

### 3.4.5 Třída JoinedReceive

Pokud potřebujeme hromadně zpracovávat položky z více portů, tak třída JoinedReceive nám může být velmi nápomocná. Pomocí ní docílíme toho, že se aplikační logika vyková pouze za předpokladu, že na všechny sledované porty, byla odeslána nějaká položka. Tím, že se logika vykoná až po příjmu položek na všech portech, tak nezáleží na pořadí jejich odeslání, a tím se lze elegantně vyhnout problémům, známých pod názvem deadlock.

Po vykonání aplikační logiky, je možné zaslat libovolnou hodnotu zpět na volající stranu, a tím zajistit komunikaci mezi klienty a serverem. Užitečná může být také možnost dynamického přiřazení počtu portů, které se mají hromadně vyhodnocovat.

```
Port doublePort = new Port<double>();
Port stringPort = new Port<string>();

// Vytvoření instance třídy JoinedReceive, která vykoná aplikační logiku
// pouze za toho předpokladu, že oba sledované porty,
// obdrží nějakou položku.
Arbiter.Activate(
    dispatcherQueue,
    doublePort.Join( // První sledovaný port.
        stringPort, // Druhý sledovaný port.
        (value, stringValue) => // Aplikační logika, která se má vykovat.
        {
            value *= 2.0d;
            stringValue = value.ToString() + value.ToString();
            // Modifikované položky budou odeslány zpět.
            doublePort.Post(value);
            stringPort.Post(stringValue);
        }
    )
);

// Odeslání dat. Nezáleží na pořadí, ve kterém budou data odeslána.
// Až po odeslání dat na všechny sledované porty,
// bude aplikační logika vykonána.
doublePort.Post(4.0d);
stringPort.Post("Výsledek: ");
```

Výpis 11: Použití třídy JoinedReceive

Uvedený příklad ve výpisu 11 ukazuje způsob, jak pomocí rozšiřující metody Join(), lze vytvořit a použít instanci třídy JoinedReceive. Ta má zde za úkol naslouchat portu doublePort a stringPort, a teprve po obdržení položek na oba porty, vykonat aplikační logiku, která předaná data modifikuje, a pak je již modifikována zašle zpět.

Alternativou ke zde použité rozšiřující metodě Join() třídy Port, by bylo použití metody Arbiter.JoinedReceive().

### 3.4.6 Třída MultipleItemReceive

Určitě je žádoucí, abychom měli možnost hromadně zpracovat prvky, přijaté jedním portem. A právě to je úkolem třídy MultipleItemReceive. Můžeme tak například počkat na požadovaný počet přijatých odpovědí, a ty pak hromadně zpracovat. Tento přístup zpracování odpovědí, je vidět ve výpisu 12.

```
// Vytvoření služby naslouchající na portu.
ServicePort servicePort = SimpleService.Create(dispatcherQueue);

// Vytvoření portu sdíleného požadavky, na který budou přicházet odpovědi.
PortSet responsePort = new PortSet<string, Exception>();

// Počet odeslaných požadavků a přijímaných odpovědí.
int count = 10;

// Odeslání požadavků.
for (int i = 0; i < count; i++)
{
    // Vytvoření požadavku.
    GetState getState = new GetState();

    // Přiřazení sdíleného portu pro odpovědi.
    getState.ResponsePort = responsePort;

    // Odeslání požadavku.
    servicePort.Post(getState);
}

// Vytvoření instance třídy MultipleItemReceive, která bude čekat
// na obdržení zadaného počtu odpovědí, které pak jako kolekci předá
// anonymní třídě k dalšímu zpracování.
Arbiter.Activate(dispatcherQueue,
    responsePort.MultipleItemReceive(
        count, // total responses expected
        (successes, failures) => Console.WriteLine("Přijato: " +
successes.Count + failures.Count)
    )
);
```

Výpis 12: Použití třídy MultipleItemReceive

Ve výše uvedeném příkladu je vidět, že se na servicePort odešle deset požadavků, kde každý z odeslaných požadavků má zaregistrovaný responsePort, na který se odešle odpověď buď jako řetězec, nebo jako výjimka.

Dále je pomocí rozšiřující metody MultipleItemReceive(), vytvořena instance třídy MultipleItemReceive, která načte výsledky odeslané na responsePort. Po načtení deseti výsledků, což je zadáno proměnnou requestCount, jsou tyto výsledky jako kolekce, předány anonymní metodě ke zpracování. Ta zobrazí počet obdržených odpovědí typu String a Exception.

Je dobré si uvědomit, že nezáleží na pořadí, ve kterém odpovědi na port responsePort přijdou, ale rozhodující je pouze jejich počet, jelikož až po dosažení požadované počtu, je vyvolána metoda, která může výsledky dále zpracovat.

### 3.4.7 Třída Interleave

Doposud jsme předpokládali, že metody volané po příchodu zprávy na port, jsou na sobě zcela nezávislé. To znamená, že mohou běžet paralelně, a jejich činnost se vzájemně nikterak neovlivňuje. V praxi se však často tento ideální stav nevyskytuje. Podívejme se například na následující metodu ve výpisu 13, která je volaná po obdržení zprávy, mající za úkol aktualizovat stav komponenty.

```
void UpdateHandler(UpdateState update)
{
    // Protože se při aktualizaci stavu sčítá řetězec obdržený
    // v příchozí zprávě s řetězcem udávající aktuální stav
    // stav komponenty, tak během vykonávání této metody
    // nesmí být vykonávaná žádná jiná, která by stav komponenty
    // změnila před dokončením manipulace s proměnnou state.
    this.state = update.State + this.state;

    // Jako odpověď se vrátí hodnota proměnné state.
    update.ResponsePort.Post(this.state);
}
```

Výpis 13: Aktualizace stavu komponenty

Pro řešení této nelehké situace, nám CCR nabízí třídu Interleave. Podívejme se na modifikaci metody ve výpisu 14, inicializující komponentu pomocí výše zmíněné třídy.

```
private void Initialize()
{
    // Aktivuje instanci třídy Interleave, která umožní definovat
    // způsob volání metod reagujících na příchozí zprávy tak,
    // aby byl brán ohled na metody reagující na předchozí zprávy.
    Arbiter.Activate(dispatcherQueue,
        Arbiter.Interleave(
        new TeardownReceiverGroup(
            // Metoda StopHandler bude vykonaná pouze jednou,
            // až se dokončí metody UpdateHandler() a GetStateHandler().
            // Po skončení metody StopHandler(),
            // bude z portu odregistrována.
            Arbiter.Receive<Stop>(false, servicePort, StopHandler)
        ),
        new ExclusiveReceiverGroup(
            // Metoda UpdateHandler() se zavolá pouze tehdy,
            // pokud se zrovna nevykonává žádná jiná metoda
            // UpdateHandler(), GetStateHandler() nebo StopHandler().
            Arbiter.Receive<UpdateState>(true, servicePort, UpdateHandler)
        ),
        new ConcurrentReceiverGroup(
            // Metoda GetStateHandler() bude moci běžet paralelně sama
            // se sebou, ale nikdy nepoběží paralelně
            // s metodou UpdateHandler() nebo StopHandler().
            Arbiter.Receive<GetState>(true, servicePort, GetStateHandler)
        ))
    );
}
```

Výpis 14: Použití třídy Interleave

Jak tedy můžeme vidět, tak se nyní metoda UpdateHandler() spustí pouze za předpokladu, že zrovna neběží žádná jiná metoda UpdateHandler(), GetStateHandler() nebo StopHandler(), která by mohla manipulovat s proměnnou state.

# 4 Decentralized Software Services

Decentralized Software Services [4] lze chápat jako nadstavbu Concurrency and Coordination Runtime. Zjednodušeně se dá říci, že díky DSS se CCR stává nezávislým na jednom konkrétním stroji.

## 4.1 Úvod do DSS

V DSS jsou jednotlivé služby pojaty jako zdroje, které jsou dostupné jak pro kód klientské aplikace, tak i pro uživatelské rozhranní, využívané například při simulaci robotů. Tím že logika napsaná pomocí DSS vystupuje jako služba, kterou lze volat pomocí síťové komunikace i z jiného počítače, než na kterém samotná služba běží, tak není pomocí DSS větší problém, vytvářet vysoce škálovatelné distribuované systémy.

Hlavním cílem DSS je umožnit co nejjednodušeji navrhnout a realizovat robustní aplikace. Vhodné použití nalezne zejména tam, kde lze aplikaci navrhnout jako samostatně běžící služby, které navzájem mezi sebou komunikují. K této komunikaci DSS používá Decentralized Software Services Protocol, což je komunikační protokol založený na protokolu SOAP.

Běhové prostředí DSS nám nabízí monitorování, logování, zabezpečení a mnoho dalšího, a to nezávisle na tom, zda celá aplikace běží na jednom či více počítačích. Služby lze psát nejenom ve vývojovém nástroji Visual Studio, ale lze k tomu použít i Microsoft Visual Programming Language. Použití VPL je od klasického programování rozdílné především tím, že se požadované služby skládají z jiných, pouhým jejich přetažením na plátno a jejich pospojováním podle toku dat do požadovaného logického celku.

## 4.2 Problémové oblasti řešené pomocí DSS

Nyní si přibližme tři problémové oblasti, které se DSS snaží řešit.

### 4.2.1 Robustnost

Během používání každé aplikace, dochází k chybám způsobených buďto daty, se kterými aplikace neumí správně naložit, nebo k chybám, které mohou být zapříčiněny dočasnou nedostupností nějakého zdroje, nebo chybou v kódu. A právě to, jak se umí aplikace s těmito chybami vypořádat, označujeme jako robustnost aplikace. DSS se snaží o snížení škod zapříčiněných chybami tím, že data před odesláním nějaké službě klonuje, a tím jsou v případě neúspěšného vykonání služby k dispozici v nezměněné podobě. Lze tak službu volat opakovaně, a teprve při několikátém neúspěchu, se může informace o chybě reportovat nadřazené službě.

### 4.2.2 Škálovatelnost

Aby aplikace mohly být robustní, je potřeba aplikační logiku rozdělit do několika komponent, a tak selhání zapříčiněné jednou komponentou, nemusí ohrozit činnost ostatních komponent. To jaké komponenty budou v konkrétní aplikaci použity, jaké mají na sebe navzájem vazby a omezení, označujeme jako škálovatelnost aplikace. DSS na rozdíl od většiny jiných architektonických přístupů, umožňuje skládat celkovou aplikaci z jednotlivých služeb až za běhu, což je umožněno tím, že informace o vazbách a omezení služeb na jiné, jsou součástí komunikačního protokolu Decentralized Software Services Protocol.

### 4.2.3 Sledovatelnost

Snad v každé rozsáhlejší aplikaci je potřeba mít přehled o tom, v jakém stavu se jednotlivé komponenty aplikace nacházejí, s jakými konkrétními daty se právě pracuje, jestli někde nedošlo k nějakému chybovému stavu, atd. DSS řeší zpřístupnění těchto všech důležitých informací tím, že každá služba je přístupná pomocí konkrétního URI, které slouží pro přístup ke všem důležitým informacím o službě, a to včetně ke stavům, ve kterých se služba nachází, nebo k datům, které jsou předávány během komunikace mezi jednotlivými službami. Ke zjištění těchto informací lze použít libovolný webový prohlížeč, nebo přístup k nim lze přímo integrovat do jiných aplikací.

## 4.3 Základy DSS

### 4.3.1 Jednotlivé části DSS

Základním stavebním blokem v Distributed Software Services, jak již z názvu je patrné, jsou služby. Pod těmito službami si lze představit hardware jako například senzory, pohonné jednotky, software jako například uživatelské rozhranní, úložiště dat, a mnoho dalšího.

Služby pro svůj běh využívají uzly DSS. Pod pojmem DSS uzel si lze představit běhové prostředí, které službám umožňuje jejich vytvoření, spuštění a řízení a to až do té doby, než je služba z DSS uzlu odstraněna, nebo je zastavena činnost samotného DSS uzlu. Velmi přínosnou podporou je to, že DSS umožňuje službám komunikovat mezi sebou navzájem bez ohledu na tom, zda jsou jednotlivé služby umístěny v jednom či více uzlech.

Nyní si stručně popíšeme jednotlivé části, ze kterých se skládá každá DSS služba.

#### 4.3.1.1 Identifikátor služby

Když DSS uzel vytváří instanci nějaké služby, tak se pomocí konstruktoru, dynamicky přiřazuje Universal Resource Identifier (URI). A právě tento identifikátor je využíván při komunikaci mezi jednotlivými službami. Pokud se chceme dozvědět informace, které služba o sobě zveřejňuje, tak stačí zadat toto URI do webového prohlížeče.

Konkrétní identifikátor služby, může být zadán jedním z následujících způsobů:

1. Identifikátor služby může být uvedeno pomocí DSS Service Manifest.
2. Identifikátor služby může být zaslán pomocí Create požadavku.
3. Identifikátor může být uveden přímo v kódu služby, pomocí atributu ServicePort.

#### 4.3.1.2 Popis služby

Popis služby slouží k popisu jejího chování. Tento popis je dostupný opět pomocí URI, které je automaticky vygenerováno při vytvoření nového projektu služby, k čemuž lze použít nástroj DSS New Service Generation Tool (DssNewService.exe).

#### 4.3.1.3 Stav služby

Pomocí stavu služby můžeme zjistit její stav v libovolném časovém bodu. Stav služby můžeme chápat jako jakýsi dokument, který popisuje aktuální obsah služby. Například u služby, která by zabezpečovala chod motoru, by oním stavem mohl být dokument, informující o aktuální teplotě motoru, otáčkách, stavu oleje či benzínu.

#### 4.3.1.4 Partnerské služby

Aby služby mohly být seskupovány do větších celků, které se pak navenek tváří jako jedna služba, musí mít služba možnost identifikovat, které další služby na ní závisí, nebo naopak, na kterých službách je závislá. Partnerské služby se uvádějí pomocí atributů, které mohou mít mnoho upřesňujících parametrů. Lze tak definovat, zda partnerská služba musí být vytvořena spolu se službou, která na ni závisí, nebo mohou být předány parametry, které se mohou použít při vytváření instance partnerské služby.

#### 4.3.1.5 Hlavní port

Data jsou službami přijímána pomocí hlavního portu. Jedná se o stejný port, jaký jsme již poznali v části, která se věnovala Concurrency and Coordination Runtime. Jediným způsobem, jak si služby mohou mezi sebou předávat data, je tedy řešeno zasíláním zpráv na hlavní port. V kódu se port označuje atributem ServicePort. Použití tohoto atributu, může vypadat například jako ve výpisu 15:

```
[ServicePort("/TestSample")]
private TestSampleOperations mainPort = new TestSampleOperations();
```

Výpis 15: Použití atributu ServicePort

To, jaké typy zpráv může daný port přijímat, je v tomto případě definováno třídou TestSampleOperations. Služba musí přijímat zprávy definované protokolem Decentralized Software Service Protocol, nebo Hypertext Transfer Protocol. DSSP vyžaduje podporu zpráv, jejichž doručení spouští operace LOOKUP, DROP, GET a REPLACE. U HTTP je vyžadována podpora pouze dva typy zpráv, které odpovídají operacím GET a POST. Pokud chceme mít službu, která bude podporovat oba dva protokoly, tak lze port definovat například způsobem, zobrazeným ve výpisu 16:

```
public class TestSampleOperations : PortSet<DsspDefaultLookup, DsspDefaultDrop,
Get, Replace, HttpGet, HttpPost>
{
}
```

Výpis 16: Příklad definice portu služby

#### 4.3.1.6 Ovladače služby

Pro každou portem podporovanou operaci, musí mít služba zaregistrován ovladač, který bude zpracovávat příchozí zprávy. Jedinou výjimkou jsou operace vyvolávané zprávami typu DsspDefaultLookup a DsspDefaultDrop, pro které v DSS existuje výchozí implementace. Registrace ovladače se provádí pomocí atributu ServiceHandler. Například pro operaci GET, by registrace mohla být jako ve výpisu 17:

```
[ServiceHandler]
public IEnumerator<ITask> GetHandler(Get get)
{ get.ResponsePort.Post(this.State); yield break; }
```

Výpis 17: Registrace ovladače služby

Logika umístěná do ovladače služby, může komunikovat s ostatními službami jedním ze dvou způsobů:

1. Zaslat zprávu jako požadavek na jinou službu.
2. Vyvolat událost informující o změně stavu služby, kterou mohou jiné služby zachytit.

#### 4.3.1.7 Události

Dobrým způsobem, jak mohou být služby informovány o změnách probíhajících v nějaké jiné službě, je použití událostí. Každá služba může vyvolávat události, které informují o změně stavu služby.

Protože stav se může změnit po příchodu nějaké zprávy na hlavní port, tak informace o změně stavu je odeslána na lokální port služeb, které si přejí být o změně stavu informovány, a tento lokální port musí přijímat stejný typ zpráv, který vedl ke změně stavu sledované služby.

### 4.3.2 Datový model DSS

Stav služby reprezentuje službu v daném časovém okamžiku. Na stav se lze dívat jako na jakýsi dokument, který službu popisuje. O tom, co z čeho bude tento dokument složen, se rozhoduje použitím atributu DataContractAttribute, DataMemberAttribute a DataMemberConstructorAttribute, ve zdrojovém kódu služby DSS.

#### 4.3.2.1 DataContractAttribute

U názvu tříd, struktur a výčtů, indikuje použití atributu DataContractAttribute to, že daná struktura se má stát nositelem informací, mezi jednotlivými službami.

#### 4.3.2.2 DataMemberAttribute

Tímto atributem se označují konkrétní proměnné a vlastnosti, jejichž hodnoty se mají mezi službami přenášet.

#### 4.3.2.3 DataMemberConstructorAttribute

Pokud k nastavení počátečních hodnot, je potřeba použít konstruktor, tak se použije atribut DataMemberConstructorAttribute v kombinaci s atributem DataContractAttribute. Pokud nějaká vlastnost nebo proměnná ale nemá být daným konstruktorem ovlivněna, tak se u požadované vlastnosti nebo proměnné, použije atribut DataMemberConstructorAttribute s hodnotou -1, v kombinaci s atributem DataMemberAttribute.

## 4.4 Ukázka vytvoření služby

Nastal čas ukázat si, jak se dá DSS služba vytvořit. Po instalaci Microsoft Robotics Developer Studia, jsou k dispozici příklady na adrese C:\Users\<jméno>\Microsoft Robotics Dev Studio 2008 R3\samples\ServiceTutorials\

### 4.4.1 Vygenerování skeletu a kompilace služby

K vygenerování skeletonu služby, se dá použít nástroj DssNewService, který se spouští pomocí příkazového řádku DSS Command Prompt. Tento příkazový řádek se nachází v nabídce Start / All Programs / Microsoft Robotics Developer Studio 2008 R3 / DSS Command Prompt. Po jeho spuštění, přejdeme příkazem „cd Samples“ do složky s příklady. Protože se ve složce již nějaké příklady vyskytují, tak si pro náš projekt zvolíme nějaký unikátní název, například TestService1, a ten zadáme jako parametr nástroji DssNewService. Vše ilustrují následující řádky, ve výpisu 18:

```
cd Samples
dssnewservice /namespace:Robotics /service:TestService1
```

Výpis 18: Použití nástroje DssNewService

Nyní přejdeme do nově vytvořené složky, a pomocí příkazu „dir“ si necháme zobrazit vygenerované složky a soubory. Vidíme, že mezi soubory je i TestService1.csproj, což je projektový soubor pro Visual Studio, které spustíme příkazem „start TestService1.csproj“, jak je vidět ve výpisu 19.

```
cd TestService1
dir
start TestService1.csproj
```

Výpis 19: Spuštění projektu TestService1

Po otevření projektu ve vývojovém nástroji Visual Studio, stiskneme klávesu F6, a tím se překompilují zdrojové soubory. V souboru **TestService1Types.cs**, se nachází několik tříd.

Třída Contract obsahuje řetězcovou konstantu Identifier, která musí být unikátní, neboť identifikuje konkrétní službu. V podstatě je možné zadat libovolný řetězec, u kterého bude zaručeno, že je unikátní.

Třída TestService1State může obsahovat libovolný počet vlastností, sloužících k definici stavu služby. Tyto vlastnosti se označí atributem DataMember, což zajistí jejich serializovatelnost, která je potřebná pro přenos hodnot vlastností po síti.

Třída TestService1Operations definuje hlavní port služby, na který budou doručovány příchozí zprávy.

Třída Get slouží k přenosu stavu služby. Každá služba má třídu Get rozdílnou z toho důvodu, že může obsahovat jiný název třídy, sloužící k reprezentaci stavu služby. Jak již víme, tak v našem příkladu se jedná o třídu TestService1State.

V souboru **TestService1.cs**, je umístěna definice samotné služby, která je v našem případě tvořena třídou TestService1Service, odvozenou od třídy DsspServiceBase.

Proměnná _state je typu TestService1State, a umožňuje ostatním službám zjišťovat a v případě implementace operace Replace i nastavovat stav služby.

Proměnná _mainPort je typu TestService1Operations. Jedná se o hlavní port služby, na který budou přicházet požadavky z jiných služeb. Atributem ServicePort se definuje adresa služby a nastavuje příznak AllowMultipleInstances, který určuje, zda je dovoleno vytvářet více instancí služby.

### 4.4.2 Spuštění služby

Překompilované soubory se nacházejí ve složce bin, která je o dvě úrovně výše, než se nyní nacházíme. Přejdeme tedy do ní příkazem „cd ..\..\bin“. Po zadání příkazu „dir /P“ vidíme, že složka obsahuje velké množství souborů, a mezi nimi i takové, jako ve výpisu 20:

```
cd ..\..\bin
dir /P
...
TestService1.Y2011.M03.dll
TestService1.Y2011.M03.pdb
TestService1.Y2011.M03.Proxy.dll
TestService1.Y2011.M03.Proxy.pdb
TestService1.Y2011.M03.Proxy.xml
TestService1.Y2011.M03.Transform.dll
TestService1.Y2011.M03.Transform.pdb
...
```

Výpis 20: Zobrazení souborů po kompilaci projektu TestService1

Nástroj DssNewService přidává do názvu souborů příponu s rokem a měsícem, kdy byla služba vygenerována. Proto budete mít jinou příponu, než zde uvedenou „Y2011.M03“.

Pro samotný běh služby, je potřeba mít spuštěn DSS uzel. Při jeho spuštění pomocí příkazu „DssHost“, lze jako parametr zadat službu nebo služby, které mají v daném uzlu běžet. Jsou celkem tři způsoby, jak předat informaci o službách, které chceme v daném uzlu provozovat.

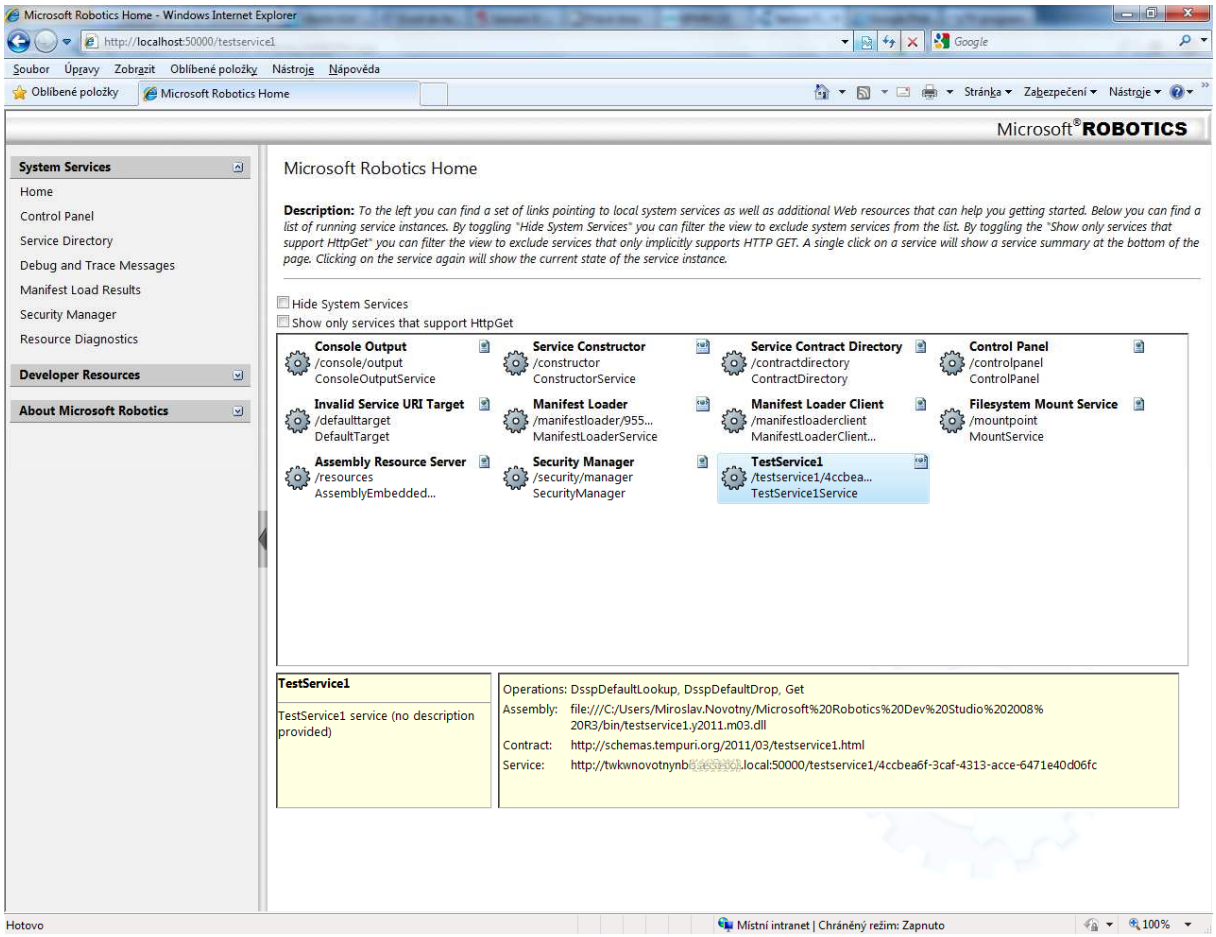
- Pomocí manifestu, použitím parametru „/manifest“.
- Pomocí assembly, použitím parametru „/dll“.
- Pomocí popisu služby, použitím parametru „/contract“.

Vyberme si například možnost využívající manifest, což je xml soubor vygenerovaný nástrojem DssNewService, a který obsahuje všechny informace potřebné k běhu služby.

```
dsshost /port:50000
/manifest:"..\samples\TestService1\TestService1.manifest.xml"
```

Výpis 21: Spuštění služby testservice1

Po spuštění DSS uzlu s naší službou, což lze provést jako ve výpisu 21, si můžeme její dostupnost otestovat tak, že si otevřeme webový prohlížeč, a zadáme adresu http://localhost:50000/testservice1, jako na obrázku 1.




Obrázek 1: Otestování spuštěné služby testservice1

Z obrázku 1 je patrné, že služba zveřejňuje svůj aktuální stav jiným službám, na adrese: http://localhost:50000/testservice1/4ccbea6f-3caf-4313-acce-6471e40d06fc.

Pokud chceme činnost DSS uzlu ukončit, tak přejdeme zpět do příkazového řádku, a stiskneme kombinaci kláves „CTRL+C“.

### 4.4.3 Podpora HTTP GET

Protože se samozřejmě nespokojíme s pouhou kompilací a spuštěním automaticky vygenerovaných souborů, tak si nyní ukážeme, jak pozměnit vygenerované soubory tak, aby služba pomocí operace HTTP GET, zveřejňovala svůj stav pomocí XML serializace stavu objektů.

V souboru „TestService1Types.cs“ jsou třídy, které slouží k popisu služby, jejího stavu, podporovaných operací a typů přijímaných či odesílaných zpráv. Jako první co musíme udělat, je tak jako ve výpisu 22, doplnit do této třídy odkaz na jmenný prostor „Microsoft.Dss.Core.DsspHttp“, který obsahuje definici zpráv, které mohou být zasílány jako odpovědi na požadavky, odesílány z nějakého HTTP klienta, jako je například webový prohlížeč.

```
using Microsoft.Dss.Core.DsspHttp;
```

Výpis 22: Doplnění jmenného prostoru Microsoft.Dss.Core.DsspHttp

Pro zobrazení stavu služby, použijeme ve třídě „TestService1State“ vlastnost, kterou nazveme třeba „Member“. Aby se mohla stát nositelem dat mezi službami, tak musí být označena atributem „DataMember“, jako ve výpisu 23.

```
[DataContract]
public class TestService1State
{
    private string member = "This is a default State.";

    [DataMember]
    public string Member
    {
        set
        {
            this.member = value;
        }
        get
        {
            return this.member;
        }
    }
}
```

Výpis 23: Třída TestService1State

Aby naše služba podporovala i zprávy typu HttpGet, tak je nutné přidat tento typ zprávy do definice kolekce portů, kterou služba používá k přijímání zpráv. Viz výpis 24.

```
[ServicePort]
public class TestService1Operations : PortSet<DsspDefaultLookup, DsspDefaultDrop,
Get, HttpGet>
{
}
```

Výpis 24: Třída TestService1Operations

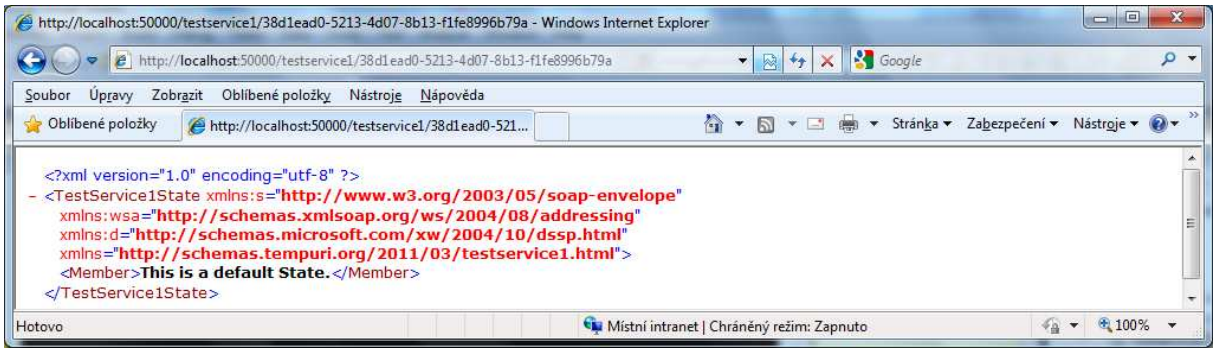
Nyní je potřeba napsat metodu, která se má zavolat po obdržení zprávy typu HttpGet. Chování služby obstarává třída „TestService1Service“, definovaná v souboru „TestService1.cs“. Do tohoto souboru tedy nejprve opět vložíme odkaz na jmenný prostor „Microsoft.Dss.Core.DsspHttp“, a pak napíšeme metodu volanou po příchodu zpráv HttpGet.

Jak vidíme ve výpisu 25, tak metoda po obdržení požadavku se správou typu HttpGet, odešle odpověď pomocí třídy HttpResponseType, která jako parametr konstruktoru přebírá objekt, který má být použit jako nositel informace. V našem případě je oním nositelem informace proměnná _state, která je instancí typu TestService1State.

```
/// <summary>
/// Po obdržení zprávy typu HttpGet, odešle odpověď s informacemi o stavu služby.
/// </summary>
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> HttpGetHandler(HttpGet httpGet)
{
    httpGet.ResponsePort.Post(new HttpResponseType(_state));
    yield break;
}
```

Výpis 25: Metoda HttpGetHandler() třídy TestService1Service

Ke kompilaci a spuštění služby, nyní stačí ve Visual Studiu stisknout klávesu F5. Zobrazí se konsole informující o průběhu spouštění služby. Povšimněme si zejména adresy služby, která je uvedená včetně jejího id. Výslednou odpověď generovanou námi vytvořenou metodou, si můžeme zobrazit pomocí webového prohlížeče, do kterého zadáme adresu, jako na obrázku 2:
`http://localhost:50000/testservice1/d4c87947-e1a8-42cf-9fa7-51372e98ddf1`



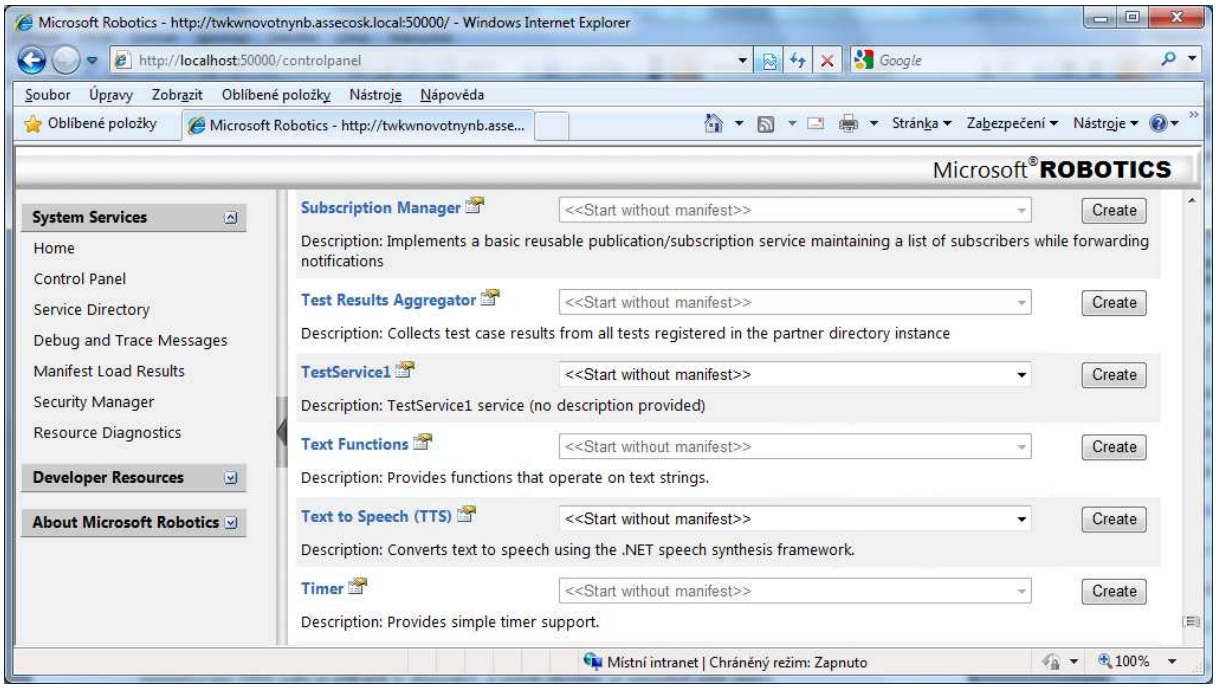

Obrázek 2: http://localhost:50000/testservice1/d4c87947-e1a8-42cf-9fa7-51372e98ddf1

### 4.4.4 Ovládací panel

Pro zjednodušení manipulace se službami, lze použít ovládací panel, což je ve své podstatě také DSS služba. Pro jeho použití, je tedy nutné nejprve spustit DSS uzel, což provedeme tak, že do DSS příkazového řádku, napíše příkaz:
`dsshost /port:50000`

Pak si otevřeme webový prohlížeč, a zadáme do něj adresu:
`http://localhost:50000/`

Tím se nám zobrazí stránka se spuštěnými DSS službami. Jak můžeme vidět, tak jich již po nastartování DSS uzlu je několik k dispozici, a jejich úkolem je usnadnit nám práci s konfigurací a během námi vytvořených služeb. Teď se ale zaměříme pouze na ovládací panel, a proto klikněte na službu nazvanou „Control Panel“. Po chvilce se nám zobrazí seznam všech dostupných služeb, jako na obrázku 3. Jistě mezi nimi naleznete i tu námi vytvořenou, která se jmenuje TestService1. Pokud se stane, že bylo nalezeno několik stejně nazvaných služeb, tak se zobrazí pod sebou.



Obrázek 3: Seznam všech dostupných služeb

Ke spuštění služby máme nyní možnost použít manifest, pokud je pro danou službu dostupný. Ten nalezne využití především v tom případě, že služba ke své činnosti potřebuje spustit i nějaké partnerské služby. V manifestu je pak uvedeno, které to jsou. Další možností je ke spuštění služby použít místo manifestu, pouze atributy použité při psaní kódu služby. V takovém případě stačí pouze kliknout na tlačítko Create. Až bude služba spuštěna, tak se pod jejím názvem zobrazí odkaz, pod kterým je dostupná. Vedle tohoto odkazu se zobrazí tlačítko Drop, který službu ukončí.

Výběrem odkazu „Service Directory“ v levém panelu, se přesuneme na stránku s přehledem spuštěných služeb. Užitečnou informací může být i přehled partnerských služeb, které jsou zobrazeny napravo od názvu samotné služby.

Tady je dobré poznamenat, že přestože pomocí ovládacího panelu můžeme jednotlivé služby spouštět a vypínat, tak při kompilaci nové verze služby, je nutné vypnout a znovu spustit celý DSS uzel.

### 4.4.5 Podpora Replace

K tomu, aby si služby mohly vzájemně nastavovat stav služby, který může ovlivnit její činnost, je vhodné podporovat operaci Replace. Ukažme si, jak do naší služby její podporu přidat.

Nejprve je potřeba do souboru TestService1Types.cs, vložit definici operace Replace (viz výpis 26).

```
/// <summary>
/// Definice operace Replace.
/// </summary>
public class Replace : Replace<TestService1State, PortSet<DefaultReplaceResponseTy
pe, Fault>>
{
}
```

Výpis 26: Definice operace Replace

Ve stejném souboru, přidáme do definice hlavního portu služby, podporu operace Replace, jako ve výpisu 27.

```
[ServicePort]
public class TestService1Operations : PortSet<DsspDefaultLookup, DsspDefaultDrop,
Get, HttpGet, Replace>
{
}
```

Výpis 27: Přidání Replace do hlavního portu

No a poslední co nám zbývá udělat, je vložit do definice třídy TestService1Service, která se nachází v souboru TestService1.cs, metodu spouštěnou po příchodu požadavku na vykování operace Replace.

```
/// <summary>
/// Po obdržení zprávy typu Replace, nahradí stav služby hodnotou,
/// která bude obsažena v příchozí zprávě.
/// </summary>
/// <param name="replace"></param>
/// <returns></returns>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> ReplaceHandler(Replace replace)
{
    _state = replace.Body;
    replace.ResponsePort.Post(DefaultReplaceResponseType.Instance);
    yield break;
}
```

Výpis 28: Definice metody ReplaceHandler()

V metodě se nahradí objekt reprezentující stav služby objektem, který je obsažen v doručené zprávě, pod vlastností Body. Potom se port volající strany zašle zpráva typu DefaultReplaceResponseType, která potvrdí úspěšné provedení operace Replace. Vše ilustruje výpis 28.

# 5 Visual Programming Language

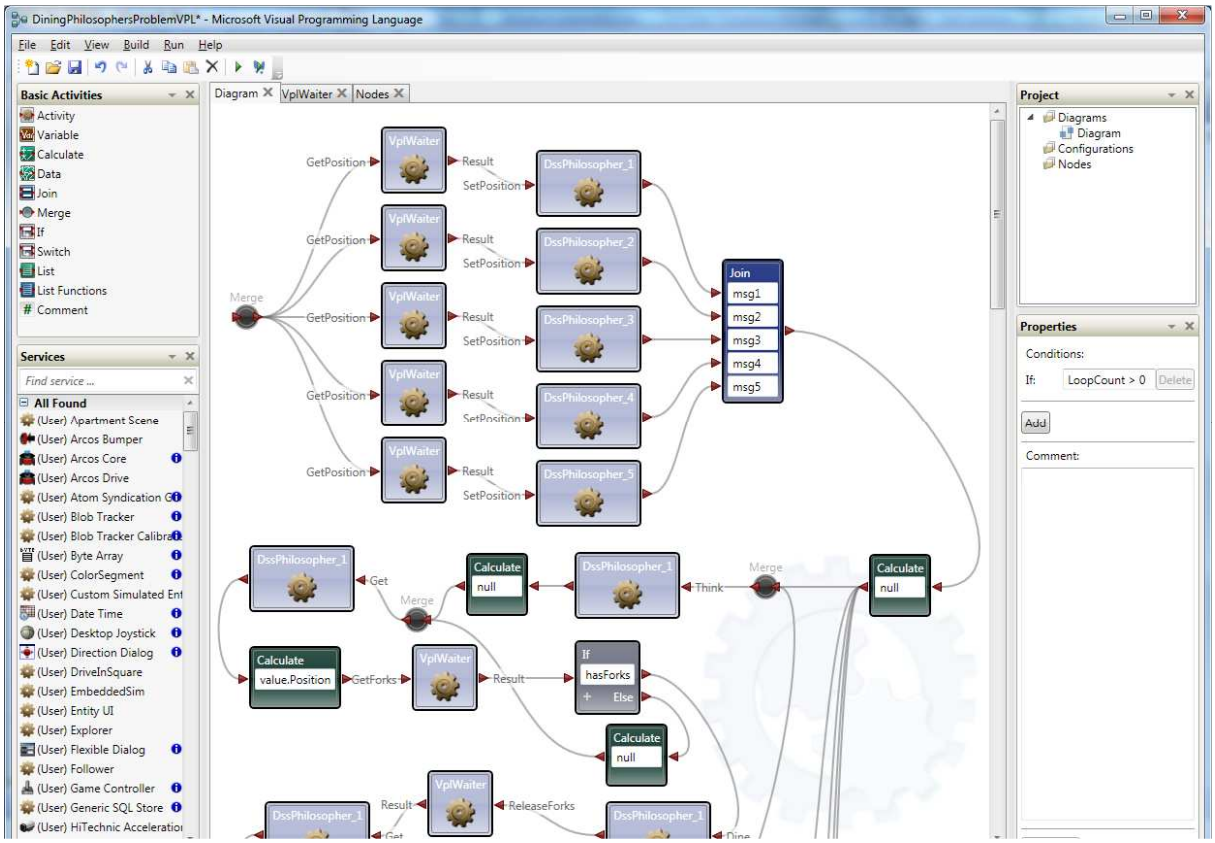
Pro tvorbu programů, nám MRDS nabízí grafický programovací jazyk, nazývaný Visual Programming Language (VPL). VPL je přínosný právě díky tomu, že lze pomocí vizuálního rozhranní, programovat technikou drag-and-drop. Přestože samozřejmě existují určité limity, tak je tato technika v mnoha případech přinejmenším zajímavá. Lze ji použít například pro rychlé vytvoření nějakého prototypového řešení. Část řešení nadefinovanou pomocí VPL, pak lze zkonvertovat do klasické C# aplikace a tu, pomocí klasického programování, dovést do finální podoby. Více naleznete přímo na stránkách firmy Microsoft [5], odkud zde uvedené informace pocházejí.

## 5.1 Uživatelské rozhranní

Visual Programming Language obsahuje vizuální nástroje, které mají za cíl co nejvíce usnadnit tvorbu výsledného programu. Ty nejčastěji používané, si nyní představíme.

### 5.1.1 Grafický editor

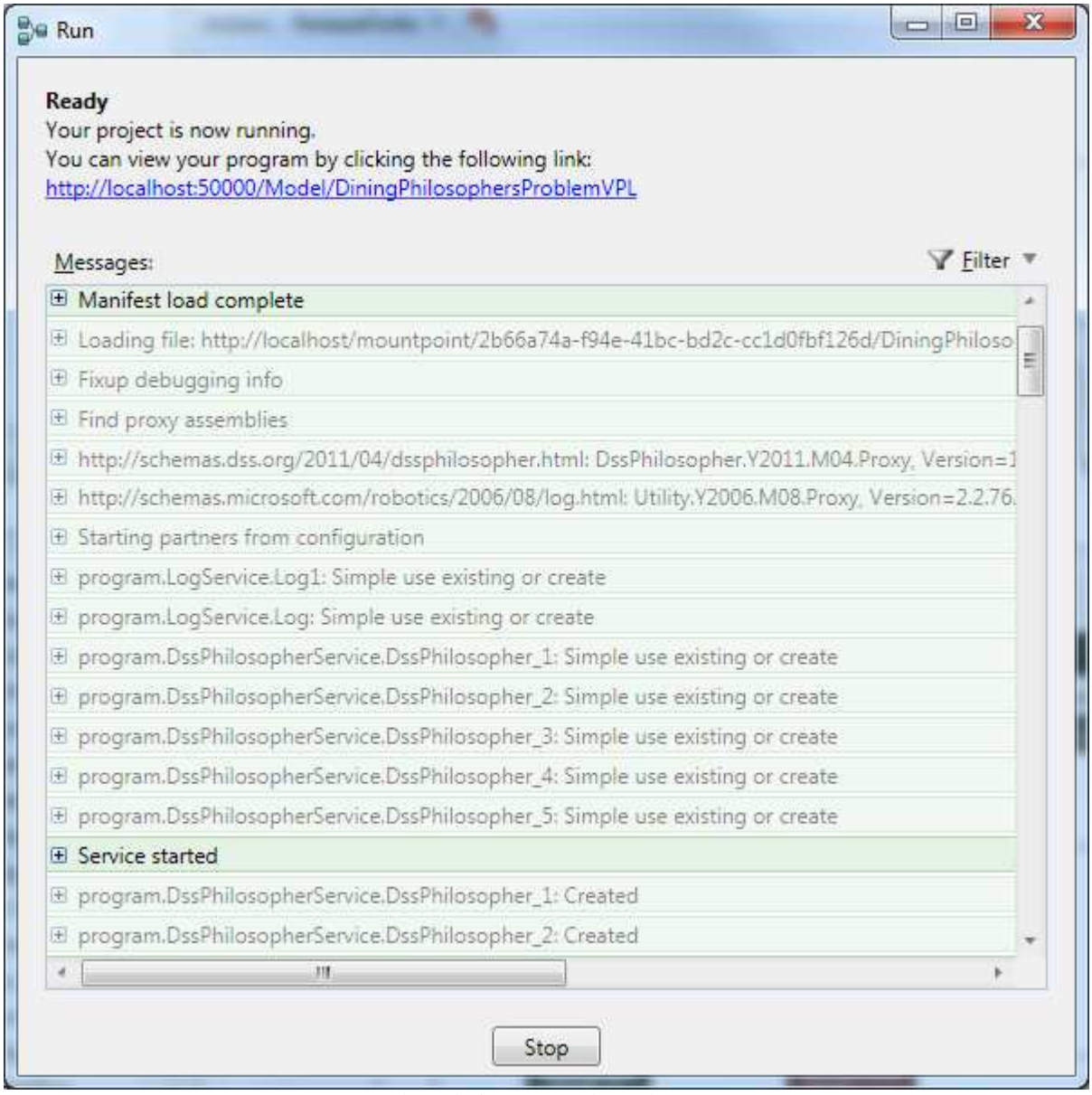
Algoritmus se realizuje pomocí grafického editoru, zobrazeného na obrázku 4, který nabízí paletu základních aktivit a paletu DSS služeb. Tyto aktivity a služby se přenášejí na plátno, kde se spojují podle požadovaného toku dat. Pokud je vhodné rozdělit celkovou logiku na části, tak lze pomocí základní aktivity nazvané příznačně Activity, vytvářet vlastní aktivity, a ty potom volat z hlavního diagramu.



Obrázek 4: Grafický editor VPL

### 5.1.2 Výstupní konsole

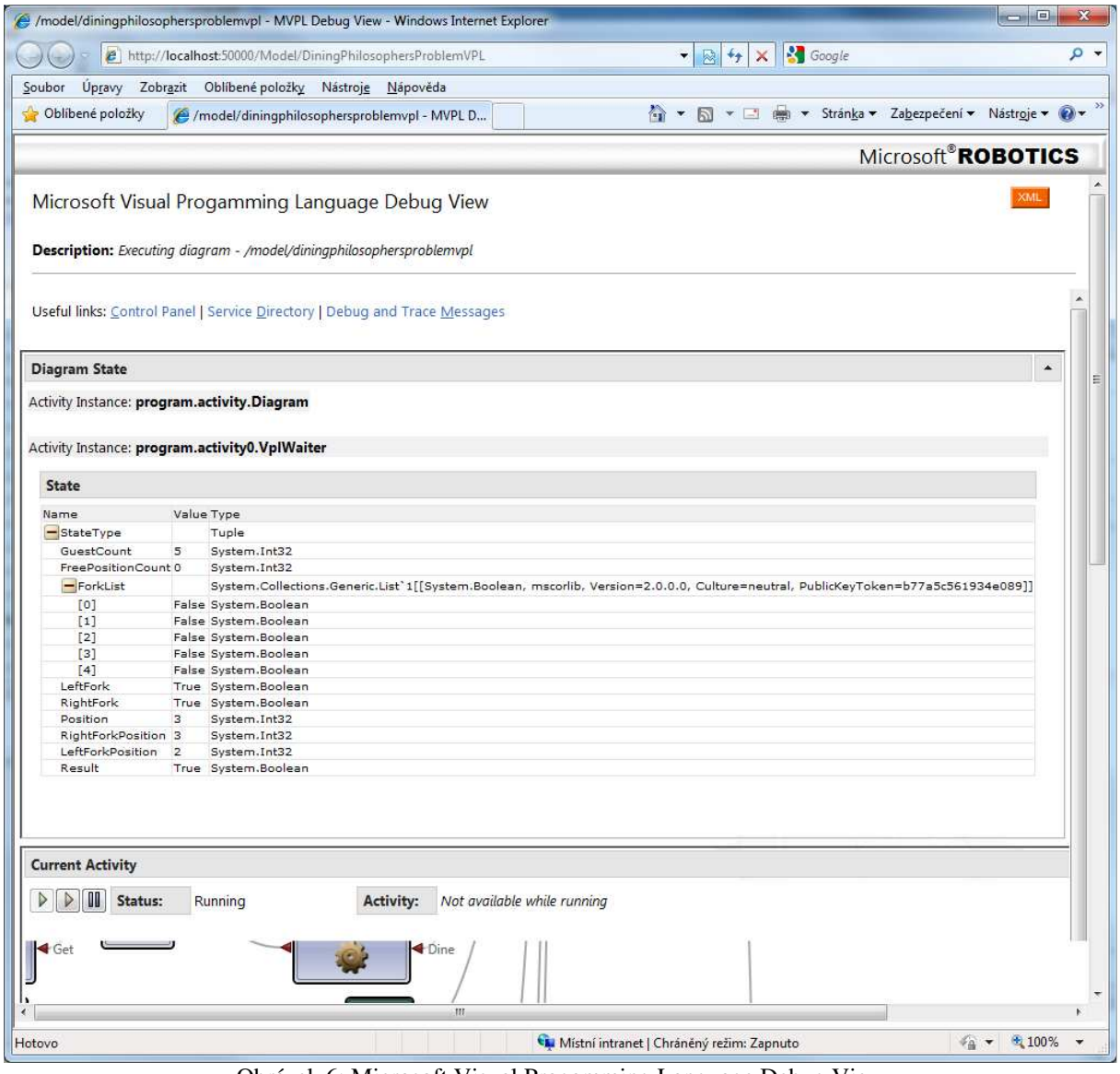
Po spuštění programu, se automaticky otevře výstupní konsole, podobná té na obrázku 5, kde lze nalézt důležité informace o průběhu programu. Chceme-li získat podrobnější informace o zalogované zprávě, tak stačí u příslušné zprávy kliknout na znak „plus“, který je umístěný před každou zprávou. Pomocí filtru je možné omezit zobrazené zprávy pouze na ty, které nás opravdu zajímají. Program lze ukončit pomocí tlačítka Stop. Zobrazený odkaz nám zobrazí stránky nazvané Microsoft Visual Progamming Language Debug View, které obsahují ještě podrobnější informace o spuštěném programu.



Obrázek 5: Výstupní konsole VPL

### 5.1.3 Microsoft Visual Progamming Language Debug View

Po spuštění programu, se zobrazí výstupní konsole s odkazem, na který pokud klikneme, tak se zobrazí internetové stránky Microsoft Visual Progamming Language Debug View. Tyto stránky obsahují spoustu důležitých informací, které lze využít především při ladění programu. Je pomocí nich totiž možné krokovat chod programu, přičemž se zobrazuje aktuální aktivita, která se právě vykonává, i s jejím stavem proměnných. Zobrazení internetové stránky Microsoft Visual Progamming Language Debug View, pak může vypadat podobně, jako na obrázku 6.



Obrázek 6: Microsoft Visual Progamming Language Debug View

## 5.2 Základní aktivity

Již v základní instalaci máme k dispozici aktivity, pomocí kterých můžeme realizovat požadovaný algoritmus. K těmto základním aktivitám je možné vytvářet služby, které budou přijímat zprávy zaslané na jejich port, a vracet hodnoty potřebné pro další řízení programu. Tyto základní aktivity a dodané služby, lze seskupovat do samostatných celků.

### 5.2.1 Activity


Pro přehlednější členění, lze seskupit libovolný počet aktivit a služeb do jedné. Takovéto aktivitě říkáme uživatelsky definovaná aktivita. Jako každá jiná aktivita, tak i uživatelsky definovaná aktivita má vstup, výstup a může vyvolávat události. Použitím uživatelsky definovaných aktivit, tak lze i složitější algoritmus zobrazit přehledně.

### 5.2.2 Calculate


Tato aktivita umožňuje vykonat základní matematické operace jako je součet, rozdíl, násobení, dělení.Také lze použít logické operátory and (&&), or (||) a not (!).

### 5.2.3 Comment


Potřebujeme-li pro lepší popis algoritmu, vložit do bloku aktivit textový komentář, tak můžeme k tomu použít aktivitu Comment.

### 5.2.4 Data


K vložení hodnot do nějaké aktivity nebo služby, jako například Variable nebo Calculate, slouží aktivita Data. Nejprve se pomocí prvkem DropDownList vybere typ dat, a pak se do textového pole zadá hodnota. Pokud je datovým typem řetězec, tak se do textového pole uvozovky nezadávají.

### 5.2.5 If


Aktivita If přesměruje vstup na první výstup za předpokladu, že je zadaná podmínka splněna. Pokud vyhodnocení podmínky je false, pak se vstup přesměruje na výstup Else.

### 5.2.6 Join


Aktivita Join kombinuje tok ze dvou vstupů. Tato funkcionalita je značně odlišná od té, kterou nabízí aktivita Merge. Zde se totiž před odesláním výstupu čeká na obdržení dat z obou vstupů. Pomocí textového pole můžeme upřesnit, které hodnoty se mají sloučit.

### 5.2.7 List


Aktivitou List, se definuje prázdný seznam datových položek. Pomocí prvku DropDownList, je potřeba vybrat příslušný datový typ. K zadání hodnot jednotlivých položek, se používá aktivita List Functions. Aby bylo možné s daty dále pracovat, tak je lze provázat s aktivitou Variable.

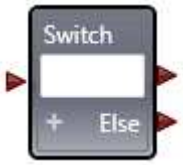
### 5.2.8 List Functions


Existující seznam, lze editovat pomocí aktivity List Function. Konkrétní funkce, která se má na seznam aplikovat, se vybere pomocí prvku DropDownList.

### 5.2.9 Merge


Aktivitou Merge dojde k jednoduchému sloučení z několika vstupů na jeden výstup. Nepoužívají se žádné podmínky, ani se nečeká na hodnoty přicházející z jiných vstupů. Účelem této aktivity je předat vstupní hodnoty z několika větví, k dalšímu zpracování.

### 5.2.10 Switch




Pokud potřebujeme zvolit určitou větev programu, na základě vyhodnocení vstupu, tak k tomu můžeme použít aktivitu Switch. Výraz určený k vyhodnocení, se vkládá do textového políčka. Další větev přidáme kliknutím na znak plus. Rozdíl oproti aktivitě If spočívá ve způsobu vyhodnocování podmínky. Zde se hledá shoda se zadanou hodnotou, které musejí mít stejný typ. U aktivity If můžeme mít pro každou větev, zcela jinou podmínku.

### 5.2.11 Variable


Aktivita Variable umožňuje zadat nebo získat hodnotu proměnné. Výběr požadované proměnné, se děje pomocí prvku DropDownList. Pokud ještě nemáme žádnou proměnnou nadefinovanou, nebo potřebujeme nadefinovat novou, tak nám k tomu postačí vybrat volbu Define Variables, nebo vybrat Variables z menu Edit. Zobrazí se nám dialog, ve kterém budeme moci zadat název proměnné a její typ. Datovým typem může být například int, double, string. Všechny použitelné typy, jsou uvedeny v tabulce 1:

| VPL Type | Description |
|---|---|
| bool | Boolean values: true, false |
| byte | 8 bit unsigned integer (0 to 255) |
| sbyte | 8 bit signed integer (-128 to 127) |
| char | character |
| decimal | fixed point decimal number (fixed precision number) |
| double | double precision (64-bit) floating point number (approx 14 significant digits) |
| float | single precision (32-bit) floating point number (approx 7 significant digits) |
| int | 32 bit signed integer |
| uint | 32 bit unsigned integer |
| long | 64 bit signed integer |
| ulong | 64 bit unsigned integer |
| short | 16 bit signed integer (-32768 to 32767) |
| ushort | 16 bit unsigned integer (0 to 65535) |
| string | character string (text) |

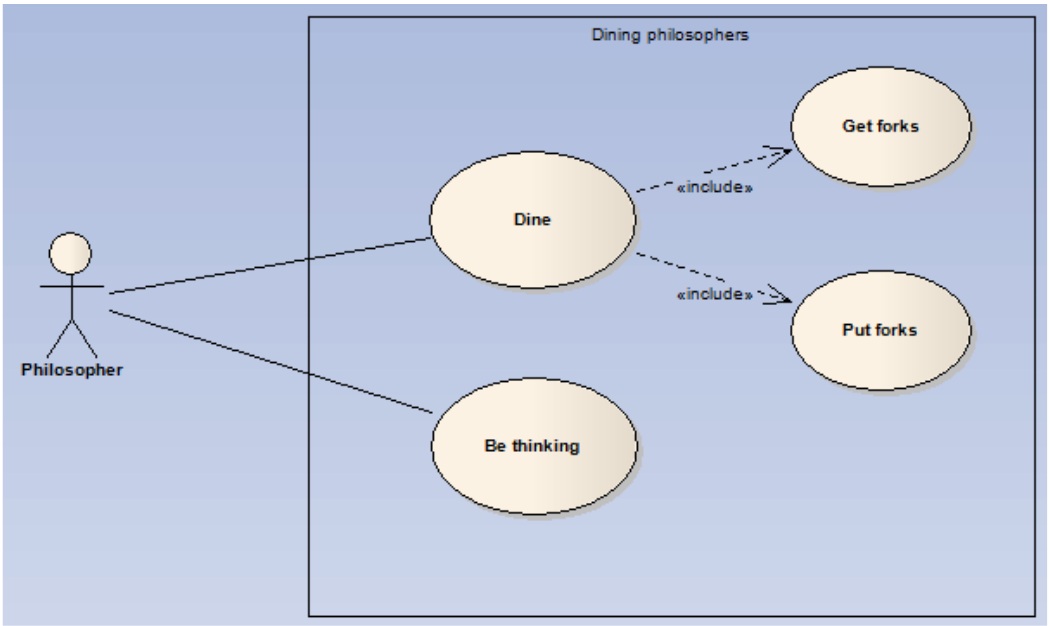
Tabulka 1: Datové typy přístupné v aktivitě Variable

# 6 Večeřící filozofové

Pravděpodobně nejznámějším zobecněným synchronizačním problémem, je problém večeřících filozofů. Nejprve si nastíníme v čem problém spočívá, ukážeme si, jak by se mohl řešit standardními prostředky platformy .NET, a pak se pokusíme tento problém vyřešit pomocí Microsoft Robotics Developer Studio. A aby to bylo ještě o něco zajímavější, tak se nezaměříme pouze na Concurrency and Coordination Runtime, Decentralized Software Services nebo na Visual Programming Language, ale zkusíme navrhnou řešení pomocí každého z nich.

Problém večeřících filozofů, jehož Use Case je na obrázku 7, a který podrobněji popisuje na svých stránkách pan Ing. Petr Olivka [6], si lze představit tak, že máme na oválném stole pět talířů se špagetami, a mezi každou dvojicí talířů leží pouze jedna vidlička. Tedy i vidliček je pět. U stolu sedí pět filozofů, každý u svého talíře. Jejich činnost spočívá v tom, že jedí nebo přemýšlejí. K tomu aby se najedli, tak musejí mít v každé ruce jednu vidličku. Protože se ale o každou vidličku dělí se svým sousedem, tak nemohou jíst všichni současně. Pokud se filozofovi podaří ze stolu získat obě vidličky, tak může jíst. Pokud ale chce přemýšlet, tak přeruší jedení a položí vidličky na stůl. V tuto chvíli mohou vidličky použít sousedící filozofové.

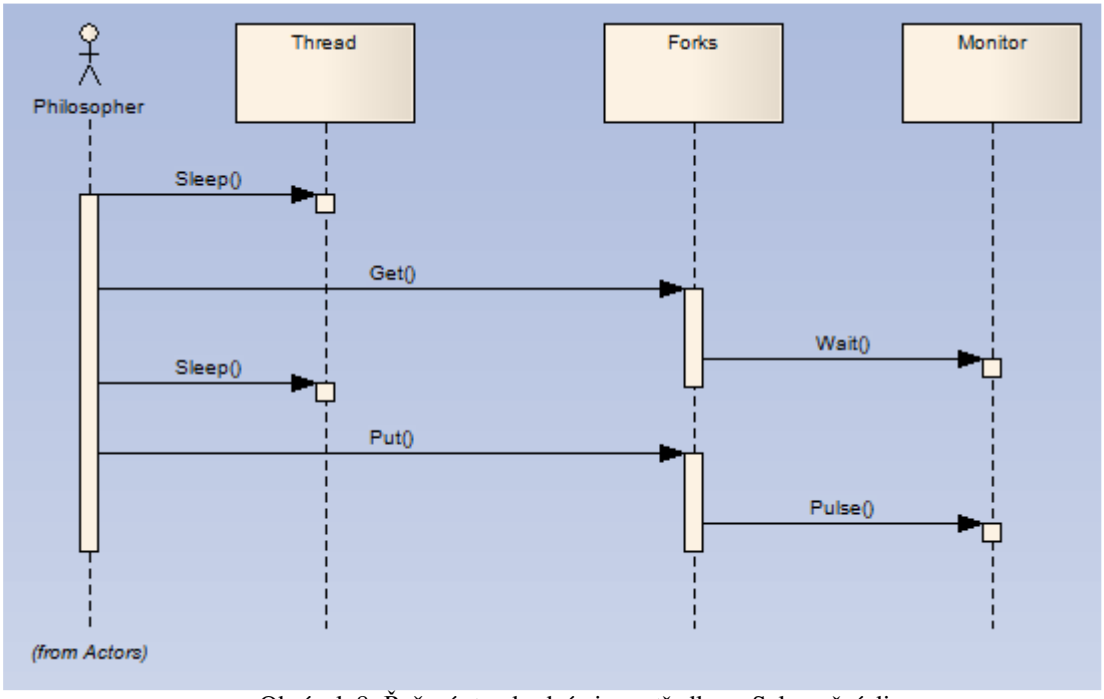
Jak tedy asi již každý tuší, tak objektem synchronizace jsou vidličky, o jejichž získání, či jejich položení na stůl, smí v jeden okamžik, usilovat pouze jeden filozof. Pokud by tomu tak nebylo, tak by se po získání jedné vidličky, mohl ve stejný okamžik té druhé zmocnit sousední filozof, a my bychom nevěděli, kdy se máme znovu pokusit o její převzetí, a co máme udělat s již získanou vidličkou.




Obrázek 7: Use Case večeřících filozofů

## 6.1 Řešení standardními prostředky

Podívejme se nejprve na způsob, kterým by šlo problém večeřících filozofů, vyřešit standardními prostředky, které nám nabízí platforma .NET. Poznamenejme, že pomocí metody Sleep(), se simuluje čas strávený přemýšlením nebo jedením. Pro lepší představu, je na obrázku 8, zobrazen sekvenční diagram. Jako základ zde uvedeného řešení, posloužily stránky Threads in C#: Dining philosophers [7].




Obrázek 8: Řešení standardními prostředky - Sekvenční diagram

### 6.1.1 Třída Forks

Třída Forks mimo jiné obsahuje metody Get() a Put(), které obsahují hlavní část logiky, řešící celý problém. Tyto metody zobrazuje výpis 29. Pomocí bloku lock() se zajistí, že všechna vlákna, která do tohoto chráněného bloku vstupují s odkazem na stejný objekt, který vystupuje v roli zámku, mohou do bloku vstupovat pouze jednotlivě. Je-li v bloku nějaké vlákno, tak ostatní se zařadí do fronty a čekají, až předchozí vlákno chráněný blok opustí. Klíčovým slovem this je určeno, že oním objektem, který má plnit roli zámku, je instance třídy Forks.

```
        /// <summary>
        /// Pokouší se získat vidličky tak dlouho, dokud se to nepodaří.
        /// </summary>
        /// <param name="left">Pořadové číslo levé vidličky.</param>
        /// <param name="right">Pořadové číslo pravé vidličky.</param>
        public void Get(int left, int right)
        {
            lock (this)
            {
                while (this.forkArray[left] || this.forkArray[right])
                {
                    Monitor.Wait(this);
                }
                this.forkArray[left] = true;
                this.forkArray[right] = true;
            }
        }

        /// <summary>
        /// Položí zadané vidličky na stůl.
        /// </summary>
        /// <param name="left">Pořadové číslo levé vidličky.</param>
        /// <param name="right">Pořadové číslo pravé vidličky.</param>
        public void Put(int left, int right)
        {
            lock (this)
            {
                this.forkArray[left] = false;
                this.forkArray[right] = false;
                Console.WriteLine("Forks number {0} and {1} were put on the table.
    ", left, right);
                Monitor.Pulse(this);
            }
        }
```

Výpis 29: Metoda Forks.Get() a Forks.Put()

Neméně zajímavé jsou metody Monitor.Wait() a Monitor.Pulse(). První jmenovaná zajistí, že pokud nejsou požadované vidličky k dispozici, tak se vlákno pozastaví, a umožní dalšímu čekajícímu vláknu, aby vstoupilo do chráněného bloku. Metoda Pulse() pozastavené vlákno probudí, a umožní mu tak opětovně zkontrolovat, zda již jsou obě vidličky dostupné.

### 6.1.2 Třída Philosopher

Konstruktor třídy Philosopher na základě pozice u stolu zjistí, které vidličky filozofovi náleží, a pak vytvoří vlákno, které po spuštění vykonává logiku umístěnou do metody Run(). Proměnná random slouží ke generování pseudonáhodných čísel, které jsou určené jako časové intervaly k uspání vlákna po dobu přemýšlení, či jedení, jak vidíme ve výpisu 30.

```
    /// <summary>
    /// Provede základní inicializaci vlastností.
    /// </summary>
    /// <param name="position">Pozice (číslo židle) filozofa u stolu.</param>
    /// <param name="forks">Všechny vidličky u stolu.</param>
    public Philosopher(int position, Forks forks)
    {
        this.Position = position;
        this.Left = position;
        this.Right = position + 1 < forks.Count ? position + 1 : 0;
        this.Forks = forks;
        Thread thread = new Thread(new ThreadStart(Run));
        thread.Name = "Philosopher " + this.Position;
        thread.Start();
    }

    /// <summary>
    /// V cyklu zajišťuje přemýšlení a jedení. Po deseti cyklech se metoda ukončí.
    /// </summary>
    public void Run()
    {
        try
        {
            Console.WriteLine("Philosopher {0} has number of left fork {1} and rig
ht fork {2}.", this.Position, this.Left, this.Right);
            for (int i = 0; i < 10; i++)
            {
                int thinkTimeout = random.Next(100, 500);
                Thread.Sleep(thinkTimeout);
                this.Forks.Get(this.Left, this.Right);
                int eatTimeout = random.Next(100, 500);
                Console.WriteLine("Philosopher {0} is eating.", this.Position);
                Thread.Sleep(eatTimeout);
                this.Forks.Put(this.Left, this.Right);
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        Console.WriteLine("Philosopher {0} was finished.", this.Position);
    }
```

Výpis 30: Konstruktor a metoda Run() třídy Philosopher

Metoda Run() začíná tím, že zobrazí informační hlášku pro identifikaci filozofa a vidliček, které bude používat. Pak se vlákno na náhodně zvolenou dobu uspí, a tím simuluje přemýšlení. Po ukončení přemýšlení, se pomocí metody Get() zařadí do fronty na vidličky. Po jejich získání se opět na náhodně zvolený časový interval uspí, a tím simuluje jedení. Až časový interval vyprší, tak vidličky uvolní. Tento cyklus se provede desetkrát.

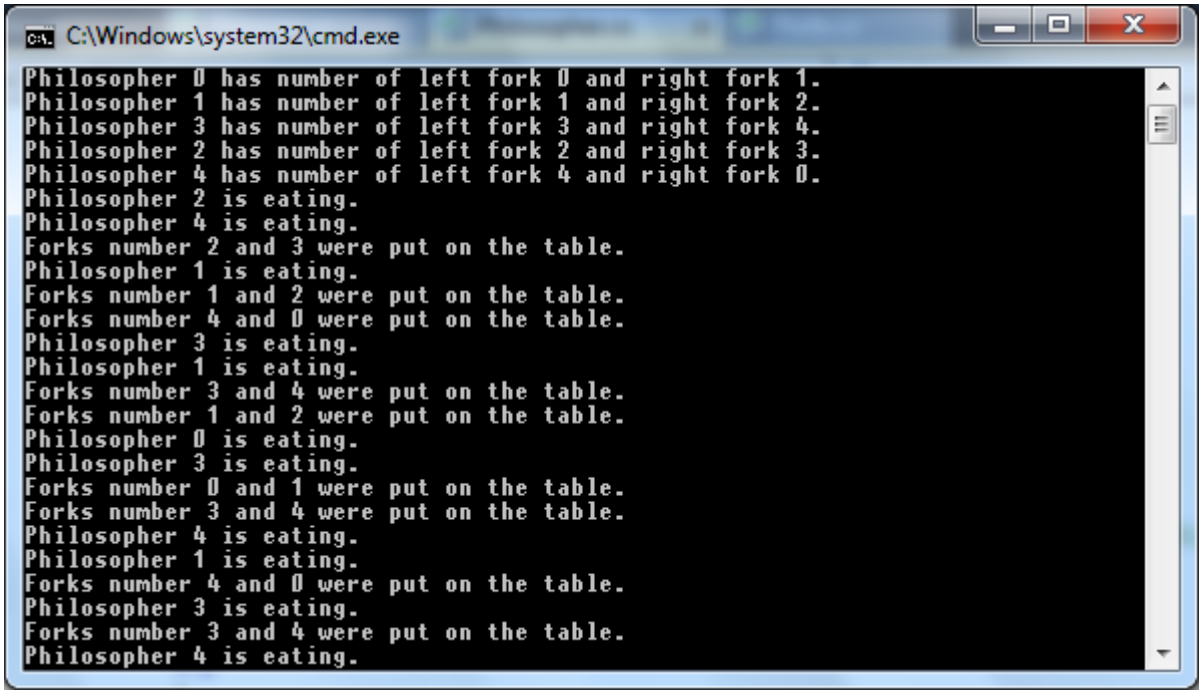
### 6.1.3 Třída Program

Třída program, jak je vidět z výpisu 31, obsahuje statickou metodou Main(), která je vstupním bodem celé aplikace.

```
class Program
{
    /// <summary>
    /// Metoda vytvoří požadovaný počet filozofů a adekvátní počet vidliček.
    /// </summary>
    /// <param name="args"></param>
    static void Main(string[] args)
    {
        // Počet filozofů a vidliček.
        int number = 5;
        // Vytvoření vidliček.
        Forks forks = new Forks(number);
        // Vytvoření filozofů.
        for (int i = 0; i < number; i++)
        {
            new Philosopher(i, forks);
        }
    }
}
```

Výpis 31: Třída Program

### 6.1.4 Výstup programu




Obrázek 9: Výstup programu

Na obrázku 9 vidíme část výstupu programu. Tento výstup se bude po každém spuštění lišit, protože vygenerované intervaly určené k simulaci přemýšlení a jedení, budou jiné. Důležité je si uvědomit, že při správné implementaci se nikdy nesmí stát, že by zároveň jedli dva sousedící filozofové. Zároveň nesmí nastat situace, že by nějaký filozof nikdy nemohl získat vidličky.

### 6.1.5 Zobrazení informací o běhu vláken

Jelikož výstup textu do konsole se někomu může jevit jako málo přesvědčivý důkaz o správné implementaci daného problému, tak použijeme ladící nástroj Profiler, který během činnosti programu nashromáždí informace o všech běžících vláknech, a tyto informace pak zobrazí v přehledných reportech. Pro náš program zobrazil report, který je vidět na obrázku 10.

**Most Contended Resources**

Resources with the highest number of total contentions

| Name | Contentions % | Contentions |
|---|---|---|
| Handle 5 | 28,57 | 12 |
| Handle 2 | 26,19 | 11 |
| Handle 1 | 26,19 | 11 |
| Handle 4 | 9,52 | 4 |
| Handle 3 | 4,76 | 2 |

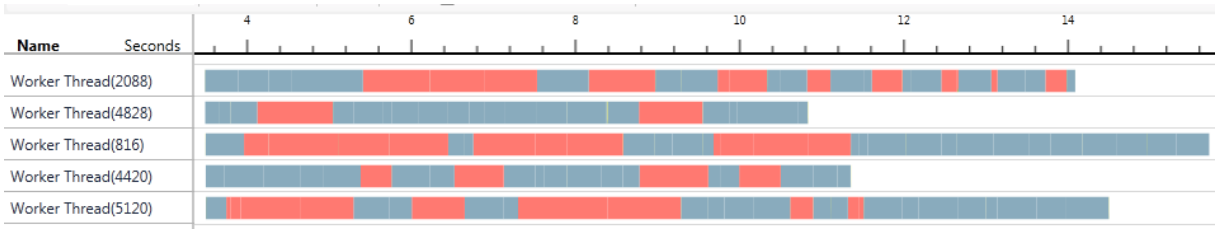
**Most Contended Threads**

Threads with the highest number of contentions

| ID | Name | Contentions % | Contentions |
|---|---|---|---|
| 816 | Philosopher 2 | 28,57 | 12 |
| 2088 | Philosopher 0 | 28,57 | 12 |
| 5120 | Philosopher 4 | 26,19 | 11 |
| 4420 | Philosopher 3 | 9,52 | 4 |
| 4828 | Philosopher 1 | 4,76 | 2 |

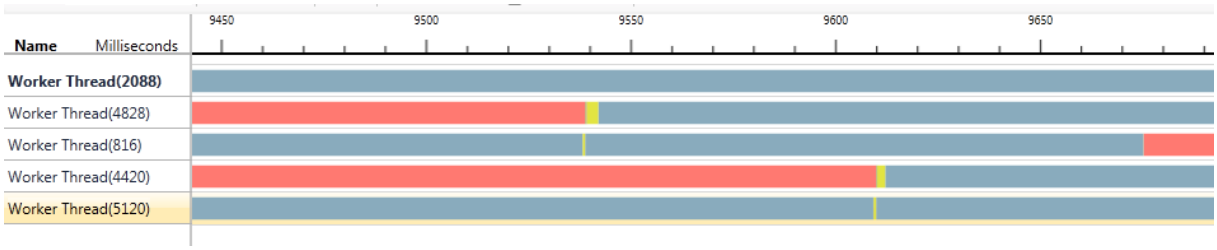
Obrázek 10: Základní report nástroje Profiler

Asi nejužitečnější informace se dají získat z následujícího grafu, zobrazeném na obrázku 11. Pomocí něj se dá lehce zjistit, jak dlouho vlákno běželo, kdy a na jak dlouho bylo uspáno, či blokováno.




Obrázek 11: Report zobrazující blokování vláken

Použijme tedy funkci zoom, a zkusme se blíže zaměřit na nějaký časový úsek. Výsledek je vidět na obrázku 12.




Obrázek 12: Report po použití funkce Zoom

Tak například na výše uvedeném obrázku vidíme, že filozof s pořadovým číslem 0, byl v zobrazeném intervalu uspán metodou Sleep(). Filozof s pořadovým číslem 1 byl synchronizován pomocí metody Wait(). Pak byl odblokován filozofem 2, vypsal informační hlášku do konsole, a následně byl uspán metodou Sleep(). Protože přešel ze synchronizace do režimu spánku, tak si můžeme být jisti tím, že se mu podařilo získat vidličky a simuluje jedení.

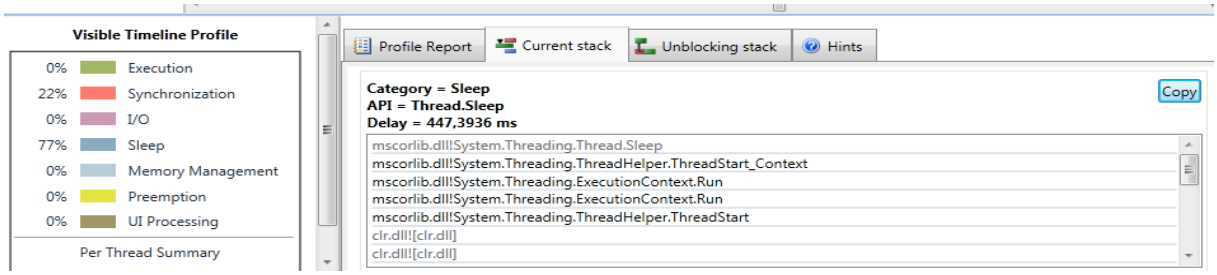
Víme, že v jeden okamžik nemohou zároveň jíst dva sousedící filozofové, a tak si nyní můžeme lehce odvodit, že filozof 0, metodou Sleep(), simuloval přemýšlení.

Filozof 2 odblokoval filozofa 1. To lze v našem programu učinit pouze metodou Pulse(), která je volaná z metody Put(). Z toho je tedy zřejmé, že filozof 2 na začátku intervalu jedl, pak vypsal hlášku informující o položení vidliček, zavolal metodu Pulse() a opět přešel do spánku metodou Sleep(), která ovšem nyní simulovala přemýšlení. Ke konci intervalu, byl po ukončení metody Sleep(), synchronizován metodou Wait(), a tedy začal vyčkávat na uvolnění vidliček.

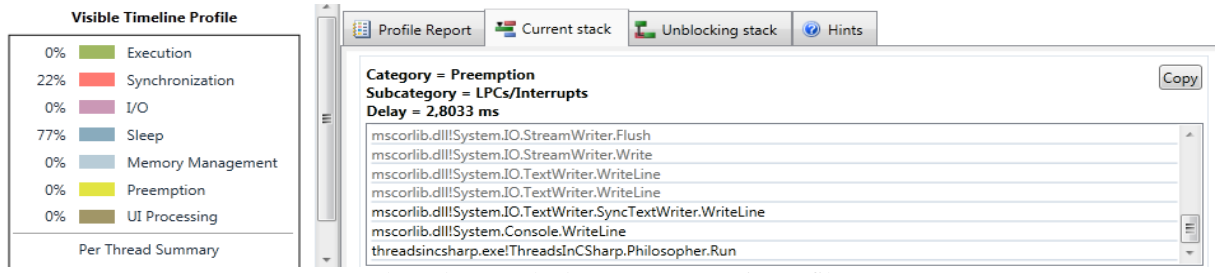
Filozof 3 stejně jako filozof 1 na začátku intervalu vyčkával na uvolnění vidliček, které se mu po odblokování filozofem 4 podařilo získat, a tak mohl, po vypsání informační hlášky, začít jíst.

Filozof 4 nejprve jedl, pak informoval o položení vidliček, probudil filozofa 3, a přešel do režimu přemýšlení.

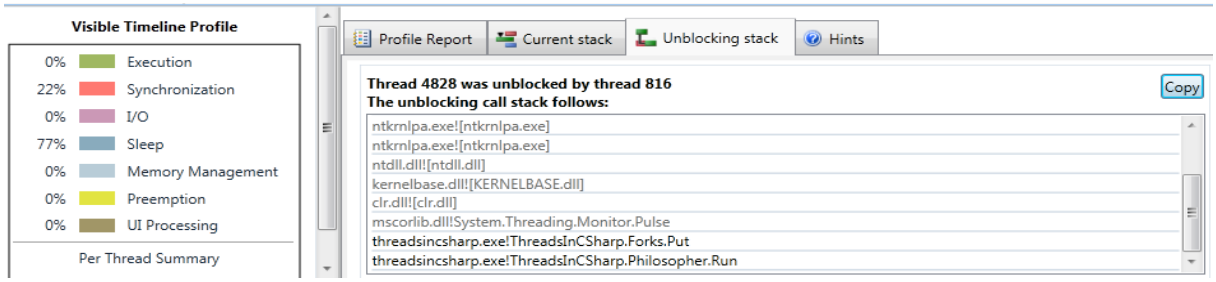
Pokud se pozastavujete nad tím, kde beru jistotu, že byla v nějaký konkrétní okamžik volaná metoda Sleep(), Wait(), posílána nějaká hláška do konsole, či došlo k odblokování nějakého filozofa metodou Pulse(), tak věřte, že se tyto informace zobrazují na záložkách reportu, po vybrání konkrétního časového intervalu, na nějakém vlákně. Pro lepší představivost, přikládám pár obrázků ze zmíněných záložek.



Obrázek 13: Záložka reportu nástroje Profiler



Obrázek 14: Záložka reportu nástroje Profiler



Obrázek 15: Záložka reportu nástroje Profiler

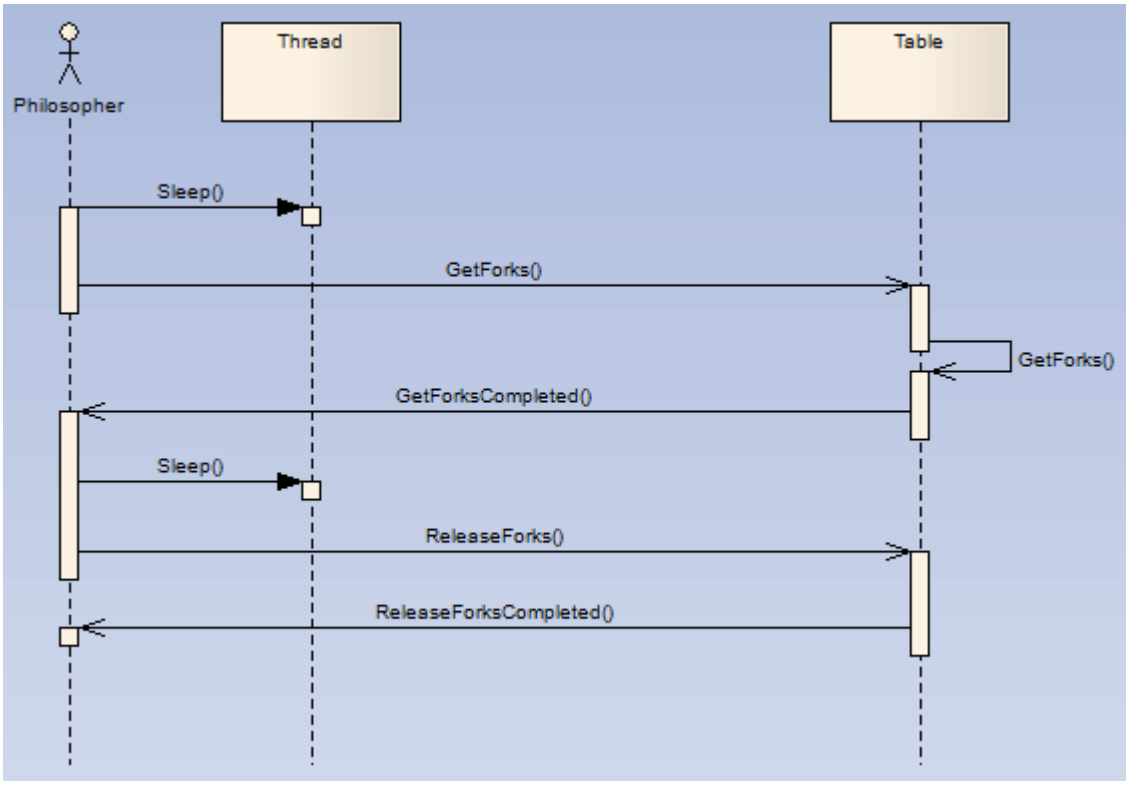
### 6.1.6 Závěr

Řešení standardními prostředky platformy .NET předpokládá, že programátor je velmi dobře obeznámen s problematikou programování aplikací aktivně využívající vláken procesu, s jejich synchronizací a dokáže správně používat metody třídy Thread i Monitor.

Přestože tyto znalosti se považují za základ programátorského umu, tak při řešení rozsáhlejších projektů, se správná synchronizace všech potřebných objektů, může stát doslova noční můrou.

## 6.2 Řešení pomocí Concurrency and Coordination Runtime

Již z názvu je patrné, že se Concurrency and Coordination Runtime zaměřuje na práci spojenou s koordinací běhu vláken. Zkusme tedy navrhnout řešení večeřících filozofů pomocí CCR. Metodu Sleep() třídy Thread, budeme používat pouze ke simulování doby přemýšlení a jedení. Vše ostatní by mělo používat prostředků, které nám CCR nabízí. Sekvenční diagram je na obrázku 16.




Obrázek 16: Sekvenční diagram řešení pomocí CCR

### 6.2.1 Pomocné třídy

Naše řešení bude využívat zasílání zpráv na porty. Proto je potřeba mít k dispozici pomocné třídy, které nám jednotlivé zprávy a porty reprezentují. V případě potřeby, nám také umožní se zprávou přenášet data, potřebná k jejímu zpracování.

```
/// <summary>
/// Zpráva informující o úspěšném přidělení vidliček.
/// </summary>
public class GetForksCompleted { }

/// <summary>
/// Zpráva informující o úspěšném uvolnění přidělených vidliček.
/// </summary>
public class ReleaseForksCompleted { }
```

Výpis 32: Třída GetForksCompleted a ReleaseForksCompleted

Tak například třídy GetForksCompleted a ReleaseForksCompleted, zobrazené ve výpisu 32, slouží k rozlišení zprávy, která je odesílána zpět na port klienta služby, který požádal o přidělení či uvolnění vidliček. Na klientovy se tak zavolá metoda, reagující na daný návratový typ třídy.

Tyto třídy se odesílají na návratový port, který bude součástí každého požadavku odeslaného službě, zajišťující manipulaci s vidličkami.

```
/// <summary>
/// Abstraktní třída pro požadavky podporované službou.
/// </summary>
public abstract class ServiceOperation
{
    public PortSet<GetForksCompleted, ReleaseForksCompleted> ResponsePort = new Po
rtSet<GetForksCompleted, ReleaseForksCompleted>();
}

/// <summary>
/// Požadavek na přidělení vidliček.
/// </summary>
public class GetForks : ServiceOperation
{
    public Philosopher Philosopher { private set; get; }

    public GetForks(Philosopher philosopher)
    {
        this.Philosopher = philosopher;
    }
}

/// <summary>
/// Požadavek na uvolnění vidliček.
/// </summary>
public class ReleaseForks : ServiceOperation
{
    public Philosopher Philosopher { private set; get; }

    public ReleaseForks(Philosopher philosopher)
    {
        this.Philosopher = philosopher;
    }
}
```

Výpis 33: Třída ServiceOperation, GetForks a ReleaseForks

Služba, jak je patrné z výpisu 33, tedy bude podporovat dva typy požadavků, GetForks a ReleaseForks. Oba dva požadavky mimo návratového portu, obsahují i vlastnost Philosopher, který může na základě svých veřejných vlastností, sloužit jako zdroj informací, potřebných ke správnému přidělení nebo uvolnění vidliček. Jak uvidíme později, tak v našem případě se bude používat vlastnost Position, udávající pozici filozofa u stolu.

Samotná služba manipulující s vidličkami, bude zprávy přijímat pomocí množiny portů, která se klientům služby, bude jevit jako jeden hlavní port. Na tento port budou klienti zasílat zprávy, které se budou rozlišovat na základě typu třídy, zaslané ve zprávě. Definici portu uvádí výpis 34.

```
/// <summary>
/// Definice portu služby.
/// </summary>
public class ServicePort : PortSet<GetForks, ReleaseForks>
{
}
```

Výpis 34: Třída ServicePort

### 6.2.2 Třída TableWithInterleave

Třída TableWithInterleave, zajišťuje samotnou manipulaci s vidličkami. Bude k tomu potřebovat několik proměnných, jak je vidno z výpisu 35, které se nastavují pomocí privátního konstruktoru.

```
/// <summary>
/// Pole vidliček. Pokud vidlička leží na stole, tak je hodnota příslušného prvku
false.
/// </summary>
private bool[] forkArray;

/// <summary>
/// Hlavní port služby, na který přicházejí požadavky.
/// </summary>
private ServicePort mainPort;

/// <summary>
/// Fronta, do které se budou vkládat úlohy, které se mají vykonat.
/// </summary>
private DispatcherQueue taskQueue;

/// <summary>
/// Provede inicializaci základních vlastností.
/// </summary>
/// <param name="taskQueue">Fronta, do které se budou vkládat úlohy, které se mají
 vykonat.</param>
/// <param name="count">Počet požadovaný vidliček. Musí se shodovat s počtem filoz
ofů.</param>
private TableWithInterleave(DispatcherQueue taskQueue, int count)
{
    this.forkArray = new bool[count];
    mainPort = new ServicePort();
    this.taskQueue = taskQueue;
}
```

Výpis 35: Proměnné a konstruktor třídy TableWithInterleave

Důvod proč je konstruktor privátní je ten, že se budeme snažit o co největší abstrakci. Třída se klientům žádajícím o přidělení vidliček nebo uvolnění vidliček již přidělených, bude jevit pouze jako port služby, na který mohou zasílat své požadavky.

```
/// <summary>
/// Vytvoří instanci třídy, a vrátí port služby.
/// </summary>
/// <param name="taskQueue">Fronta, do které se budou vkládat úlohy, které se mají
 vykonat.</param>
/// <param name="count">Počet požadovaný vidliček. Musí se shodovat s počtem filoz
ofů.</param>
/// <returns>Port služby.</returns>
public static ServicePort Create(DispatcherQueue taskQueue, int count)
{
    TableWithInterleave table = new TableWithInterleave(taskQueue, count);
    table.Initialize();
    return table.mainPort;
}
```

Výpis 36: Metoda Create()

Odkaz na tento port, lze získat po zavolání statické metody Create(). Tato metoda, jak ukazuje výpis 36, zavolá privátní konstruktor třídy a provede inicializaci hlavního portu, který je návratovým typem.

A právě tuto inicializaci hlavního portu, zobrazené ve výpisu 37, lze vnímat vzhledem k synchronizaci sdílených prostředků, které jsou v našem případě chápány jako vidličky, za klíčový moment celé aplikace.

```
/// <summary>
/// Zaregistruje ovladače příchozích požadavků.
/// </summary>
private void Initialize()
{
    // Zaregistruje ovladače příchozích požadavků.
    // Umožní nastavit chování požadované při paralelním zpracování.
    Arbiter.Activate(
        this.taskQueue,
        Arbiter.Interleave(
            new TeardownReceiverGroup(
            ),
            new ExclusiveReceiverGroup(
                Arbiter.Receive<GetForks>(true, this.mainPort, GetForksHandler),
                Arbiter.Receive<ReleaseForks>(true, this.mainPort,
ReleaseForksHandler)
            ),
            new ConcurrentReceiverGroup(
            )
        )
    );
}
```

Výpis 37: Metoda Initialize()

Inicializace služby pomocí konstrukce CCR definuje, že příchozí zprávy žádající o přidělení a uvolnění vidliček, mají být vyhodnocovány až v tom okamžiku, že se v ten samý čas nevyhodnocuje žádná zpráva doručená na hlavní port. To nám zaručí, že se během kontroly, zda je volná levá i pravá vidlička, nezmění její stav jiným vláknem, který také žádá o přidělení stejné vidličky, nebo již přidělenou vidličku vrací.

Jak vidíme z výpisu 38, tak při doručení žádosti o přidělení vidliček, bude volána metoda GetForksHandler(), při požadavku na uvolnění vidliček již přidělených, bude volána metoda ReleaseForksHandler().

```
/// <summary>
/// Ovladač požadavku na přidělení vidliček.
/// </summary>
/// <param name="getForks"></param>
private void GetForksHandler(GetForks getForks)
{
    int leftForkNumber = this.GetLeftForkNumber(getForks.Philosopher);
    int rightForkNumber = this.GetRightForkNumber(getForks.Philosopher);
    if (this.forkArray[leftForkNumber] || this.forkArray[rightForkNumber])
    {
        this.mainPort.Post(getForks);
    }
    else
    {
        this.forkArray[leftForkNumber] = true;
        this.forkArray[rightForkNumber] = true;
        getForks.ResponsePort.Post(new GetForksCompleted());
    }
}
```

Výpis 38: Metoda GetForksHandler()

Přidělení vidliček probíhá tak, že se nejprve zjistí pořadoví číslo levé a pravé vidličky, a pak se otestuje, zda jsou obě k dispozici. Pokud nejsou, tak je požadavek o jejich přidělení, poslán zpět na hlavní port služby, aby mohl být opět vyhodnocen později. Jsou-li obě vidličky volné, tak se nastaví jako zabrané, a volající straně bude pomocí návratového portu, předána zpráva o úspěšném převzetí vidliček.

Metoda volaná pro uvolnění již přidělených vidliček, vypadá tak, jak uvádí výpis 39:

```
/// <summary>
/// Ovladač požadavku na uvolnění přidělených vidliček.
/// </summary>
/// <param name="releaseForks"></param>
private void ReleaseForksHandler(ReleaseForks releaseForks)
{
    int leftForkNumber = this.GetLeftForkNumber(releaseForks.Philosopher);
    int rightForkNumber = this.GetRightForkNumber(releaseForks.Philosopher);
    this.forkArray[leftForkNumber] = false;
    this.forkArray[rightForkNumber] = false;
    Console.WriteLine("Forks number {0} and {1} were put on the table. ThreadId: {
2}", leftForkNumber, rightForkNumber, Thread.CurrentThread.ManagedThreadId);
    releaseForks.ResponsePort.Post(new ReleaseForksCompleted());
}
```

Výpis 39: Metoda ReleaseForksHandler()

Nejprve se opět zjistí pořadí levé a pravé vracející vidličky, a potom se nastaví jako volné. Do konsole se vypíše informace o jejich uvolnění, a volající straně se zašle zpráva informující o uvolnění vidliček.

Pořadové číslo levé a pravé vidličky se zjišťuje pomocí metody GetLeftForkNumber() a GetRightForkNumber(), které je jako parametr předán odkaz na objekt Philosopher, který byl zaslán jako součást zprávy žádající o manipulaci s vidličkami. Tady je dobré poznamenat, že filozof s pořadovým číslem 0, manipuluje s vidličkami 0 a 1. Poslední filozof tedy manipuluje s levou vidličkou na stejné pozici jako je on sám, ale protože všichni sedí u kulatého stolu, tak jeho pravou vidličkou je vidlička s pořadovým číslem 0. Obě metody jsou uvedeny ve výpisu 40.

```
/// <summary>
/// Na základě pozice filozofa u stolu, vrátí pořadové číslo levé vidličky.
/// </summary>
/// <param name="philosopher">Filozof, pro jehož vidličku se má pořadové číslo vrá
tit.</param>
/// <returns>Pořadové číslo levé vidličky.</returns>
private int GetLeftForkNumber(Philosopher philosopher)
{
    return philosopher.Position;
}

/// <summary>
/// Na základě pozice filozofa u stolu, vrátí pořadové číslo pravé vidličky.
/// </summary>
/// <param name="philosopher">Filozof, pro jehož vidličku se má pořadové číslo vrá
tit.</param>
/// <returns>Pořadové číslo pravé vidličky.</returns>
private int GetRightForkNumber(Philosopher philosopher)
{
    return philosopher.Position + 1 < this.forkArray.Length ? philosopher.Position
 + 1 : 0;
}
```

Výpis 40: Metoda GetLeftForkNumber() a GetRightForkNumber()

### 6.2.3 Třída Philosopher

Třída Philosopher reprezentuje filozofa, jehož činnost se v našem pojetí skládá z přemýšlení a jedení. K zajištění těchto činností, je ve třídě definováno několik proměnných a vlastností, jejichž počáteční inicializace se provádí pomocí veřejného konstruktoru.

```
/// <summary>
/// Slouží ke generování pseudonáhodných čísel.
/// </summary>
private static Random random = new Random();

/// <summary>
/// Počet opakování, po kterém se činnost filozofa ukončí.
/// </summary>
private int loopCount = 10;

/// <summary>
/// Fronta, do které se budou vkládat úlohy, které se mají vykonat.
/// </summary>
private DispatcherQueue TaskQueue { set; get; }

/// <summary>
/// Pozice (číslo židle) filozofa u stolu.
/// </summary>
public int Position { private set; get; }

/// <summary>
/// Port služby zajišťující manipulaci s vidličkami.
/// </summary>
private ServicePort TablePort { set; get; }

/// <summary>
/// Provede základní inicializaci vlastností.
/// </summary>
/// <param name="taskQueue">Fronta, do které se budou vkládat úlohy, které se mají
 vykonat.</param>
/// <param name="tablePort">Port služby zajišťující manipulaci s vidličkami.</para
m>
/// <param name="position">Pozice (číslo židle) filozofa u stolu.</param>
public Philosopher(DispatcherQueue taskQueue, ServicePort tablePort, int position)
{
    this.TaskQueue = taskQueue;
    this.Position = position;
    this.TablePort = tablePort;
}
```

Výpis 41: Proměnné a konstruktor třídy Philosopher

Ve výpisu 41 si povšimněme proměnné loopCount, pomocí které je definováno, že činnost přemýšlení a jedení, bude provedena celkem desetkrát. Vlastnost Position určuje pozici filozofa u stolu. Vlastnost TablePort je hlavním portem službu, na který budou zasílány zprávy žádající o manipulaci s vidličkami.

Celý proces přemýšlení a jedení je spouštěn veřejnou metodou Think(). Tato metoda, zobrazená výpisem 42, si nejprve vygeneruje pseudonáhodné číslo, které použije jako časový interval ke simulaci doby strávené přemýšlením. Po uplynutí této doby, bude zavolána metoda GetForksAsync(), která zahájí proces jedení. Tuto metodu můžete vidět ve výpisu 43.

```
/// <summary>
/// Simuluje přemýšlení po náhodně vygenerovaný časový interval,
/// a potom odešle požadavek na přidělení vidliček.
/// </summary>
public void Think()
{
    int thinkTimeout = random.Next(100, 500);
    Thread.Sleep(thinkTimeout);
    this.GetForksAsync();
}
```

Výpis 42: Metoda Think()

Aby se filozof mohl najíst, tak nejprve musí získat vidličky.

```
/// <summary>
/// Odešle požadavek na přidělení vidliček.
/// </summary>
private void GetForksAsync()
{
    GetForks getForks = new GetForks(this);
    Arbiter.Activate(
        this.TaskQueue,
        Arbiter.Receive<GetForksCompleted>(false, getForks.ResponsePort, GetForksC
ompletedHandler)
    );
    this.TablePort.Post(getForks);
}
```

Výpis 43: Metoda GetForksAsync()

Protože přidělení vidliček se bude dít asynchronně, tak se nejprve zaregistruje metoda GetForksCompletedHandler(), která se zavolá po obdržení zprávy informující o přidělení vidliček, která bude zaslána na návratový port. Pak se odešle požadavek o přidělení vidliček na port služby, která manipulaci s vidličkami zajišťuje. My už víme, že touto službou je instance třídy TableWithInterleave.

Po přidělení vidliček, může činnost filozofa pokračovat, a to konkrétně pomocí metody GetForksCompletedHandler(), z výpisu 44.

```
/// <summary>
/// Tato metoda je volána po úspěšném přidělení vidliček.
/// Simuluje jedení po náhodně vygenerovaný časový interval,
/// a potom odešle požadavek na uvolnění přidělených vidliček.
/// </summary>
/// <param name="getForksCompleted"></param>
private void GetForksCompletedHandler(GetForksCompleted getForksCompleted)
{
    Console.WriteLine("Philosopher {0} is eating. ThreadId: {1}", this.Position, T
hread.CurrentThread.ManagedThreadId);
    int eatTimeout = random.Next(100, 500);
    Thread.Sleep(eatTimeout);
    ReleaseForks releaseForks = new ReleaseForks(this);
    Arbiter.Activate(
        this.TaskQueue,
        Arbiter.Receive<ReleaseForksCompleted>(false, releaseForks.ResponsePort, R
eleaseForksCompletedHandler)
    );
    this.TablePort.Post(releaseForks);
}
```

Výpis 44: Metoda GetForksCompletedHandler()

Tato metoda nejprve do konsole vypíše informaci o tom, že filosof jí. Pak vygeneruje pseudonáhodné číslo, které použije jako časový interval pro simulaci doby strávené jedením. Po uplynutí této doby, zaregistruje metodu ReleaseForksCompletedHandler(), která bude zavolána po obdržení zprávy o tom, že byly vidličky uvolněny, a požádá o uvolnění vidliček, zasláním zprávy ReleaseForks.

Až k uvolnění vidliček dojde, tak bude spuštěna metoda ReleaseForksCompletedHandler(), jejíž definice je ve výpisu 45.

```
/// <summary>
/// Tato metoda je volána po úspěšném uvolnění přidělených vidliček.
/// Pokud ještě nebyl proveden požadovaný počet opakování,
/// tak znovu zahájí přemýšlení.
/// </summary>
/// <param name="releaseForksCompleted"></param>
private void ReleaseForksCompletedHandler(ReleaseForksCompleted releaseForksComple
ted)
{
    if (--loopCount > 0)
    {
        this.Think();
    }
    else
    {
        Console.WriteLine("Philosopher {0} finishes. ThreadId: {1}.", this.Positio
n, Thread.CurrentThread.ManagedThreadId);
    }
}
```

Výpis 45: Metoda ReleaseForksCompletedHandler()

Metoda nejprve zjistí, jestli se má ještě celý cyklus přemýšlení a jedení znovu opakovat, a pokud ano, tak zavolá metodu Think(). Pokud se cyklus již opakovat nemá, tak do konsole vypíše hlášku o ukončení činnosti filozofa.

### 6.2.4 Třída Program

Třída program obsahuje pouze jednu statickou metodu, zobrazenou ve výpisu 46, která slouží jako vstupní bod celé aplikace. Metoda vytvoří objekty zajišťující správu vláken a úloh, které mají být spravovanými vlákny vykonávány. Dále pomocí statické metody Create() třídy TableWithInterleave, získá odkaz na port, sloužící k zasílání zpráv žádajících o manipulaci s vidličkami. Tento port je předán všem filozofům, kteří jsou následně v cyklu vytvářeni. K zahájení činnosti filozofů, jsou volány jejich veřejné metody Think().

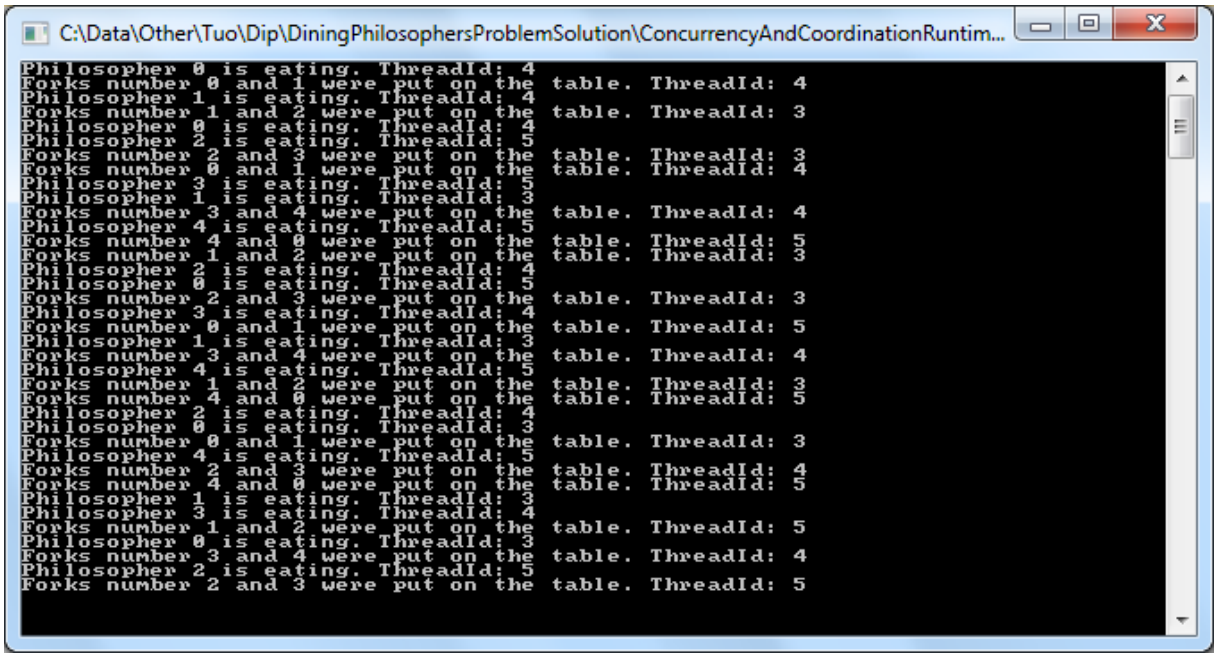
Asi nejzajímavějším okamžikem vykonávání kódu této metody, je vytváření instance třídy Dispatcher, která zajišťuje správu vláken. Po ukončení metody Main(), budou to právě tato vlákna, která budou zajišťovat činnost celého programu.

Protože jednou z výhod Concurrency and Coordination Runtime má být to, že spravovaná vlákna, pokud to není nezbytně nutné, nejsou mezi sebou blokována, tak předáme konstruktoru třídy Dispatcher požadavek na používání pouze tří vláken, přestože počet filozofů bude větší. V našem případě tedy budeme mít pět filozofů, ale pouze tři vlákna. Budeme sledovat jednotlivá vlákna podle jejich id, a tak uvidíme, jak se budou podílet na činnosti všech filozofů.

```
/// <summary>
/// Vstupní metoda programu, která vytvoří požadovaný počet filozofů.
/// </summary>
/// <param name="args"></param>
static void Main(string[] args)
{
    // Počet filozofů a vidliček.
    int number = 5;
    // Vytvoření správce vláken, kterému se v konstruktoru zadává počet požadovaný
ch vláken.
    Dispatcher dispatcher = new Dispatcher(3, "PhilosopherPool");
    // Vytvoření fronty, do které se budou vkládat úlohy, které se mají vykonat.
    DispatcherQueue taskQueue = new DispatcherQueue("PhilosopherQueue", dispatcher
);
    // Port služby zajišťující manipulaci s vidličkami.
    ServicePort tablePort = TableWithInterleave.Create(taskQueue, number);
    // Vytvoření filozofů.
    for (int i = 0; i < number; i++)
    {
        Philosopher philosopher = new Philosopher(taskQueue, tablePort, i);
        philosopher.Think();
    }
}
```

Výpis 46: Metoda Program.Main()

### 6.2.5 Výstup programu




Obrázek 17: Výpis programu

Abychom měli lepší představu o činnosti vláken, tak je jejich id zobrazováno spolu s dalšími informacemi. To je ostatně vidět na obrázku 17. Důležité je si všimnout, že skutečně nikde v jeden okamžik, nejedí dva sousedící filozofové. Zajímavé je i udávané id vlákna, které nám dokazuje, že se vlákna neblokují více, než je to nezbytně nutné, a pokud jej nemůže využít jeden z filozofů z důvodu, že nemá volné vidličky, tak je vlákno použito pro činnost jiného filozofa.

### 6.2.6 Zobrazení informací o běhu vláken

Abychom se nespoléhali pouze na výstup z konsole, probereme podrobněji i údaje, nashromážděné nástrojem Profiler. Část základního reportu je vidět na obrázku 18. Poznamenejme, že aby se v nástroji Profiler zobrazily všechny nashromážděné informace, tak bylo nutné v reportu celkového přehledu, stisknout volbu Notification / Show All code.

**Most Contended Threads**

Threads with the highest number of contentions

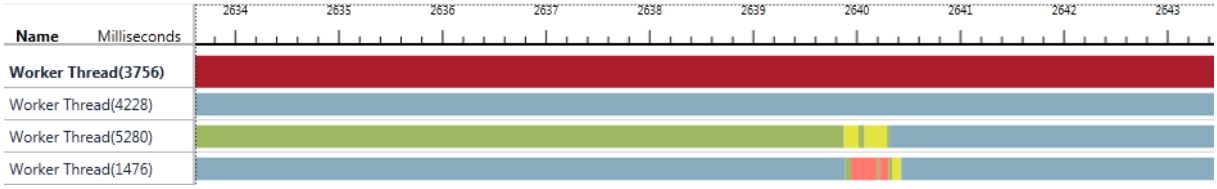
| ID | Name | Contentions % | Contentions |
|---|---|---|---|
| 4228 | PhilosopherPool | 38,46 | 10 |
| 1476 | PhilosopherPool | 34,62 | 9 |
| 5280 | PhilosopherPool | 19,23 | 5 |
| 3756 | [clr.dll] | 7,69 | 2 |

Obrázek 18: Základní report nástroje Profiler

Pro nás nejzajímavější údaje, obsahuje report Threads (viz obrázek 19).


Obrázek 19: Report Threads

Po jeho zobrazení již na první pohled vidíme, že během činnosti programu, vlákna spravovaná objektem Dispatcher, to jsou ta se jménem PhilosopherPool, trávila většinu času vykonáváním příkazu Sleep(). To pro nás jistě není nic překvapivého. Použijme tedy funkci Zoom, a přibližme si nějaký okamžik, ve kterém vlákna vykazují i jinou činnost (viz obrázek 20).




Obrázek 20: Report Threads po použití funkce Zoom

Pokud klikneme na vlákno 3756, tak se na dolní záložce zobrazí knihovny, které dané vlákno používalo. V jejich výčtu žádná knihovna microsoft.ccr.core.dll není, a tak se nebudeme tímto vláknem nadále zabývat.

Vlákno 4228 po celu dobu námi analyzovaného časového intervalu, vykonávalo metodu Sleep(). Pokud chceme vědět, zda se jednalo o simulaci přemýšlení nebo jedení, tak se musíme přesunout k časovému okamžiku, kdy vlákno do tohoto stavu přešlo. Tento okamžik nastal 2547 milisekund po startu programu. Označíme- li blok, který bezprostředně předcházel, tak na dolní záložce vidíme, že mezi volanými metodami byla metoda ConcurrencyAndCoordinationRuntime.Philosopher.GetForksCompletedHandler(), následovaná metodou System.Console.WriteLine(). Ze zdrojového kódu třídy Philosopher nyní lehce zjistíme, že se v metodě GetForksCompletedHandler() jednalo o výpis informace o jedení, po které následovala metoda Sleep(), která simulovala dobu, po kterou probíhalo jedení.
Vlákno 5280 bylo používáno knihovnou CCR k vykonávání činnosti spojené s plánováním a vykonáváním úloh. To vedlo 2640 milisekund od startu k tomu, že byla mimo jiné zavolána metoda TableWithInterleave.ReleaseForksHandler(), následované metodou System.Console.WriteLine(). Ze zdrojového kódu třídy TableWithInterleave je patrné, že se v metodě ReleaseForksHandler() zobrazovala informace o uvolnění vidliček. Následující blokem bylo vlákno opět uspáno, což bylo způsobeno voláním posloupností metod Philosopher.ReleaseForksCompletedHandler(), která jak je ze zdrojového kódu patrné, volala metodu Philosopher.Think() a ta metodou Thread.Sleep(), začala simulovat přemýšlení.

Vlákno 1476 v našem sledovaném úseku opět bylo z počátku uspané. Po přechodu na předešlý blok vidíme, že mezi volanými metodami byla posloupnost metod Philosopher.GetForksCompletedHandler(), System.Console.WriteLine(). Ze zdrojového kódu tak lehce vyčteme, že filozof informoval o tom, že právě jí, a pak tuto činnost simuloval voláním metody Thread.Sleep(). Vlákno tedy na počátku sledovaného úseku simulovalo jedení. V následovaném bloku vlákno chvilku běželo, ale profiler nemá o tomto bloku k dispozici žádné bližší informace. Pak bylo vlákno synchronizováno, pak chvilku běželo, a opět bylo synchronizováno. Z výpisu metod při synchronizaci je patrné, že vlákno v tuto dobu vykonávalo činnost spojenou s během metod tříd z knihovny microsoft.ccr.core.dll. Po uplynutí 2640 milisekund od startu programu, byla zavolána metoda Philosopher.GetForksCompletedHandler(), následovaná System.Console.WriteLine(). Vlákno tedy bylo předáno jinému filozofovi, který obdržel informaci o zajištění vidliček, vypsal informaci o tom, že jí, a pak vlákno uspal, aby tuto činnost simuloval. Právě jsme tedy byly svědky toho, že jedno vlákno dokáže pomocí CCR zajistit činnost více jak jednomu filozofovi. To že se muselo jednat o jiného filosofa je dáno tím, že na začátku sledovaného časového intervalu vlákno také simulovalo jedení, a mezi oběma těmito simulacemi nebyla předána zpráva o uvolnění vidliček. K uvolnění vidliček sice došlo pomocí vlákna 5280, ale to proběhlo v době, kdy vlákno 1476 ještě simulovalo jedení. Vlákno 5280 tedy uvolnilo vidličky jiného filozofa.

### 6.2.7 Závěr

Problém večeřících filozofů se nám pomocí Concurrency and Coordination Runtime povedl vyřešit. Potvrdilo se, že vlákna spravovaná pomocí třídy Dispatcher, nejsou nikterak zbytečně blokována. K zajištění činnosti spojené s manipulací sdílených zdrojů, kdy je nutné tuto činnost provádět pouze jedním vláknem, lze použít třídu Interleave. V situaci, kdy nelze příchozí požadavek kladně vyřídit a víme, že se situace může změnit, tak stačí požadavek poslat zpět na hlavní port. Tím se vlákno uvolní a požadavek bude znovu vyhodnocen později.

## 6.3 Řešení pomocí VPL a DSS

Decentralized Software Services jsou nástavbou Concurrency and Coordination Runtime, mající za cíl usnadnit tvorbu distribuovaných systémů, založených na vzájemném volání služeb. Visual Programming Language je programovací jazyk, založený na grafickém vyjádření toku dat. Data mohou být zpracovávána v komponentách, napsaných pomocí DSS, nebo v aktivitách, namodelovaných přímo ve VPL.
Zkusme tedy opět navrhnout řešení večeřících filozofů, tentokrát však pomocí filozofa napsaného pomocí DSS a číšníka realizovaného jako aktivitu VPL. Metodu Sleep() třídy Thread, budeme používat pouze ke simulování doby přemýšlení a jedení.

### 6.3.1 Pomocné třídy služby DssPhilosopher

Jedna z pomocných tříd pro službu filozofa, definuje stav služby. Obsahuje vlastnost udávající přidělenou pozici u stolu a počet přemýšlení a jedení, které má vykonat. Je umístěná v souboru DssPhilosopherTypes.cs. Zobrazuje ji výpis 47.

```
/// <summary>
/// Stav služby.
/// </summary>
[DataContract]
public class DssPhilosopherState
{
    /// <summary>
    /// Pozice (pořadové číslo) u stolu.
    /// </summary>
    [DataMember]
    public int Position { set; get; }

    /// <summary>
    /// Počet opakování, po kterém se činnost filozofa ukončí.
    /// </summary>
    [DataMember]
    public int LoopCount { set; get; }
}
```

Výpis 47: Třída DssPhilosopherState

Dále je definován hlavní port služby. Ten je vidět ve výpisu 48. Mimo standardních zpráv, přijímá i požadavky na nastavení přidělení pozice u stolu, simulaci přemýšlení a jedení.

```
/// <summary>
/// Hlavní port služby.
/// </summary>
[ServicePort]
public class DssPhilosopherOperations : PortSet<DsspDefaultLookup, DsspDefaultDrop
, Get, SetPosition, Think, Dine>
{
}
```

Výpis 48: Třída DssPhilosopherOperations

Zprávy pro nastavení přidělené pozice u stolu, simulaci přemýšlení a jedení, jsou definovány pomocí tříd SetPosition, Think a Dine. Každá zpráva obsahuje objekt upřesňující požadavek. Pouze požadavek pro nastavení pozice však tuto možnost využívá tím, že obsahuje vlastnost Position. To je k nahlédnutí ve výpisu 49.

```
/// <summary>
/// Zpráva pro nastavení pozice filozofa u stolu.
/// </summary>
public class SetPosition : Update<SetPositionRequest, PortSet<DefaultUpdateRespons
eType, Fault>>
{
    public SetPosition()
        : base(new SetPositionRequest())
    {
    }
}
/// <summary>
/// Požadavek zaslaný zprávou pro nastavení pozice filozofa.
/// </summary>
[DataContract]
public class SetPositionRequest
{
    /// <summary>
    /// Pozice (pořadové číslo) u stolu.
    /// </summary>
    [DataMember]
    public int Position { set; get; }
}
/// <summary>
/// Zpráva pro simulaci přemýšlení.
/// </summary>
public class Think : Update<ThinkRequest, PortSet<int, Fault>>
{
    public Think()
        : base(new ThinkRequest())
    {
    }
}
/// <summary>
/// Požadavek zaslaný zprávou pro simulaci přemýšlení.
/// </summary>
[DataContract]
public class ThinkRequest
{
}
/// <summary>
/// Zpráva pro simulaci jedení.
/// </summary>
public class Dine : Update<DineRequest, PortSet<int, Fault>>
{
    public Dine()
        : base(new DineRequest())
    {
    }
}
/// <summary>
/// Požadavek zaslaný zprávou pro simulaci jedení.
/// </summary>
[DataContract]
public class DineRequest
{
}
```

Výpis 49: Definice zpráv v souboru DssPhilosopherTypes.cs

### 6.3.2 Třída DssPhilosopherService

Třída DssPhilosopherService definuje službu, reprezentujícího filozofa. V metodě Start() se nastavuje počet přemýšlení a jedení (viz výpis 50).

```
/// <summary>
/// Při startu služby, nastaví počet opakování.
/// </summary>
protected override void Start()
{
    state.LoopCount = 10;
    base.Start();
}
```

Výpis 50: Metoda Start()

Požadavek na uložení přidělené pozice u stolu, je zpracován metodou SetPositionHandler(), jak můžeme shlédnou ve výpisu 51.

```
/// <summary>
/// Nastaví přiřazenou pozici u stolu.
/// </summary>
/// <param name="setPosition"></param>
/// <returns></returns>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> SetPositionHandler(SetPosition setPosition)
{
    state.Position = setPosition.Body.Position;
    LogInfo("SetPosition: " + state.Position);
    setPosition.ResponsePort.Post(DefaultUpdateResponseType.Instance);
    yield break;
}
```

Výpis 51: Metoda SetPositionHandler()

Jak vidíme ve výpisu 52, požadavek na přemýšlení, je vyřízen pomocí metody ThinkHandler(). Nejprve se vygeneruje časový interval, po který se následně vlákno uspí. Po vypršení intervalu se odešle volající straně informace o pozici filozofa u stolu pro případ, že by byla potřeba pro další tok dat.

```
/// <summary>
/// Simuluje přemýšlení.
/// </summary>
/// <param name="think"></param>
/// <returns></returns>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> ThinkHandler(Think think)
{
    int eatTimeout = random.Next(100, 500);
    Thread.Sleep(eatTimeout);
    think.ResponsePort.Post(state.Position);
    yield break;
}
```

Výpis 52: Metoda ThinkHandler()

Pokud na hlavní port služby reprezentujícího filozofa, dorazí požadavek na jedení, tak se zavolá metoda DineHandler(), z výpisu 53. Ta nejprve vypíše informační hlášku do konsole, a pak uspí vlákno na náhodně vygenerovaný časový interval. Potom sníží hodnotu vlastnosti, udávající počet opakování cyklu přemýšlení a jedení. Nakonec je volající straně odeslána informace o pozici filozofa u stolu.

```
/// <summary>
/// Simuluje jedení.
/// </summary>
/// <param name="dine"></param>
/// <returns></returns>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> DineHandler(Dine dine)
{
    this.LogInfo(string.Format("DineHandler, Philosopher {0}, ProcessId: {1}, Thre
adId: {2}", this.state.Position, Process.GetCurrentProcess().Id, Thread.CurrentThr
ead.ManagedThreadId));
    int eatTimeout = random.Next(100, 500);
    Thread.Sleep(eatTimeout);
    state.LoopCount--;
    dine.ResponsePort.Post(state.Position);
    yield break;
}
```

Výpis 53: Metoda DineHandler()

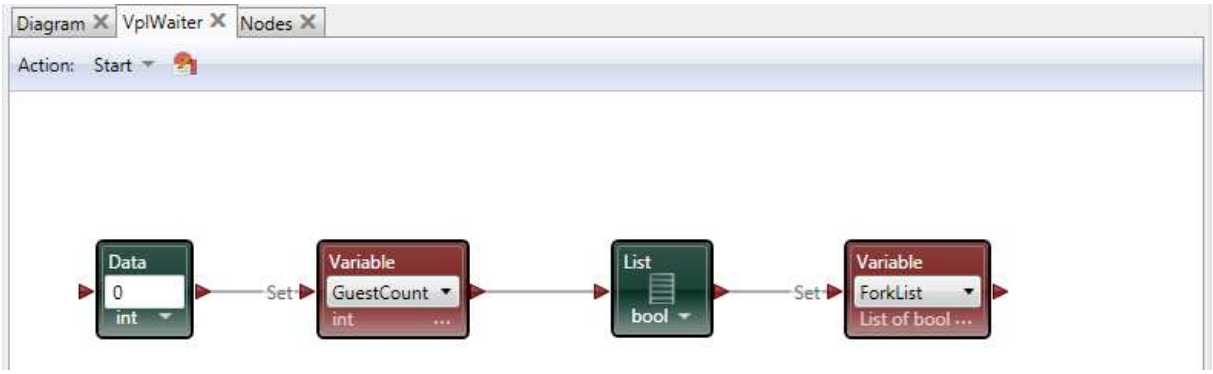
Velmi důležitý je i atribut ActivationSettings, použitý u deklarace názvu třídy, protože pomocí tohoto atributu můžeme ovlivnit počet vláken, které budou jednotlivé služby mít k dispozici a také to, jestli mohou být mezi sebou sdílená. My použijeme nastavení ShareDispatcher = false a ExecutionUnitsPerDispatcher = 1.

### 6.3.3 Aktivita VplWaiter

Číšníka zajišťující manipulaci s vidličkami u stolu, realizujeme jako VPL aktivitu. Tato aktivita se bude skládat z několika akcí, které se pak budou moci k manipulaci s aktivitou použít.

#### 6.3.3.1 Akce Start

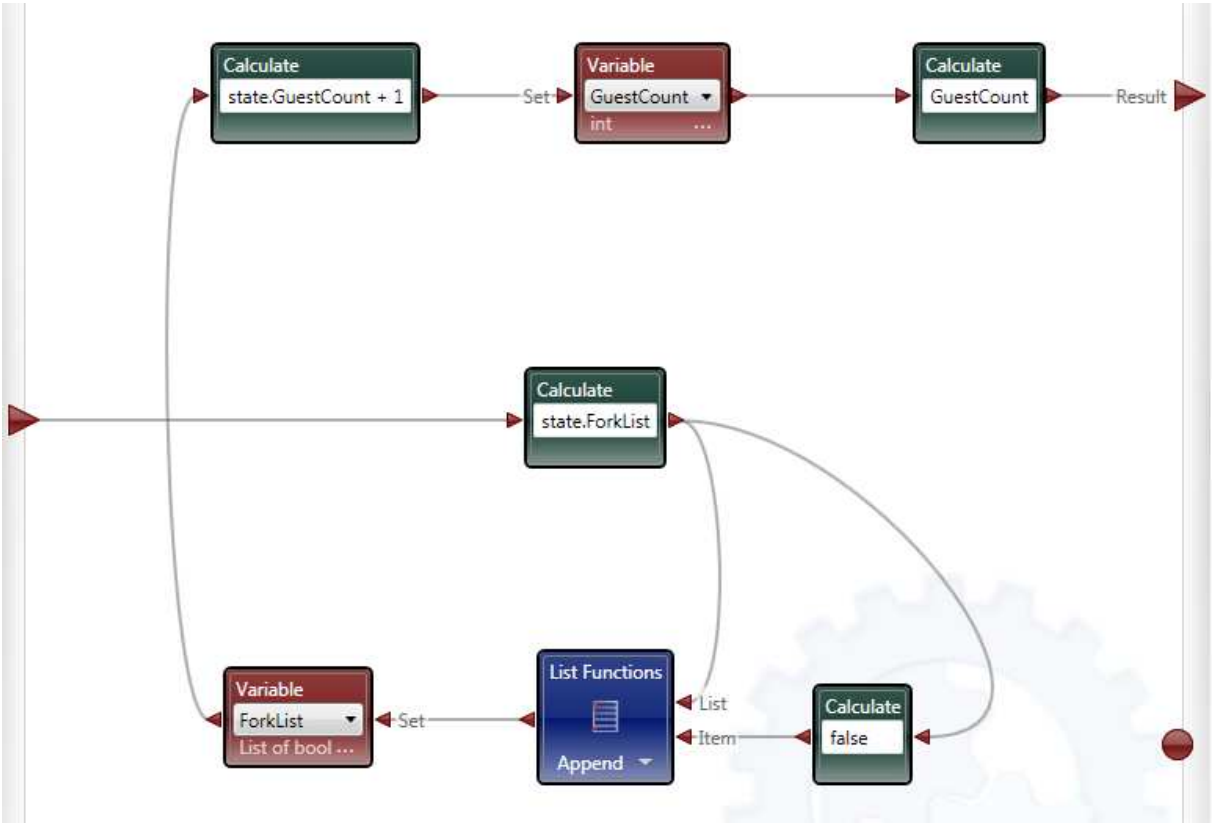
Při prvním spuštění aktivity, se zavolá akce Start, která nastaví proměnnou udávající počet již obsazených pozic u stolu na nulu, a proměnnou udávající stav vidliček, definuje jako seznam hodnot typu bool, kde false bude znamenat, že vidlička na dané pozici je volná. Definici ukazuje obrázek 21.




Obrázek 21: Akce Start

#### 6.3.3.2 Akce GetPosition

Akce GetPosition, zobrazená na obrázku 22, má za úkol přiřadit příchozímu filozofovi pozici u stolu a přidat do seznamů vidliček adekvátní počet vidliček. Připomeňme, že celkový počet vidliček má být shodný s počtem filozofů u stolu. Při každém volání akce GetPosition, se tedy pomocí funkce Append, přidá do seznamu vidliček nová položka. Potom se zvýší hodnota proměnné GuestCount, udávající celkový počet filozofů u stolu. Tento nový počet je nakonec vrácen jako výsledek akce GetPosition. Každý filozof tak tuto akci smí volat pouze jednou. První filozof volající akci GetPosition, tedy bude mít pořadové číslo 1.

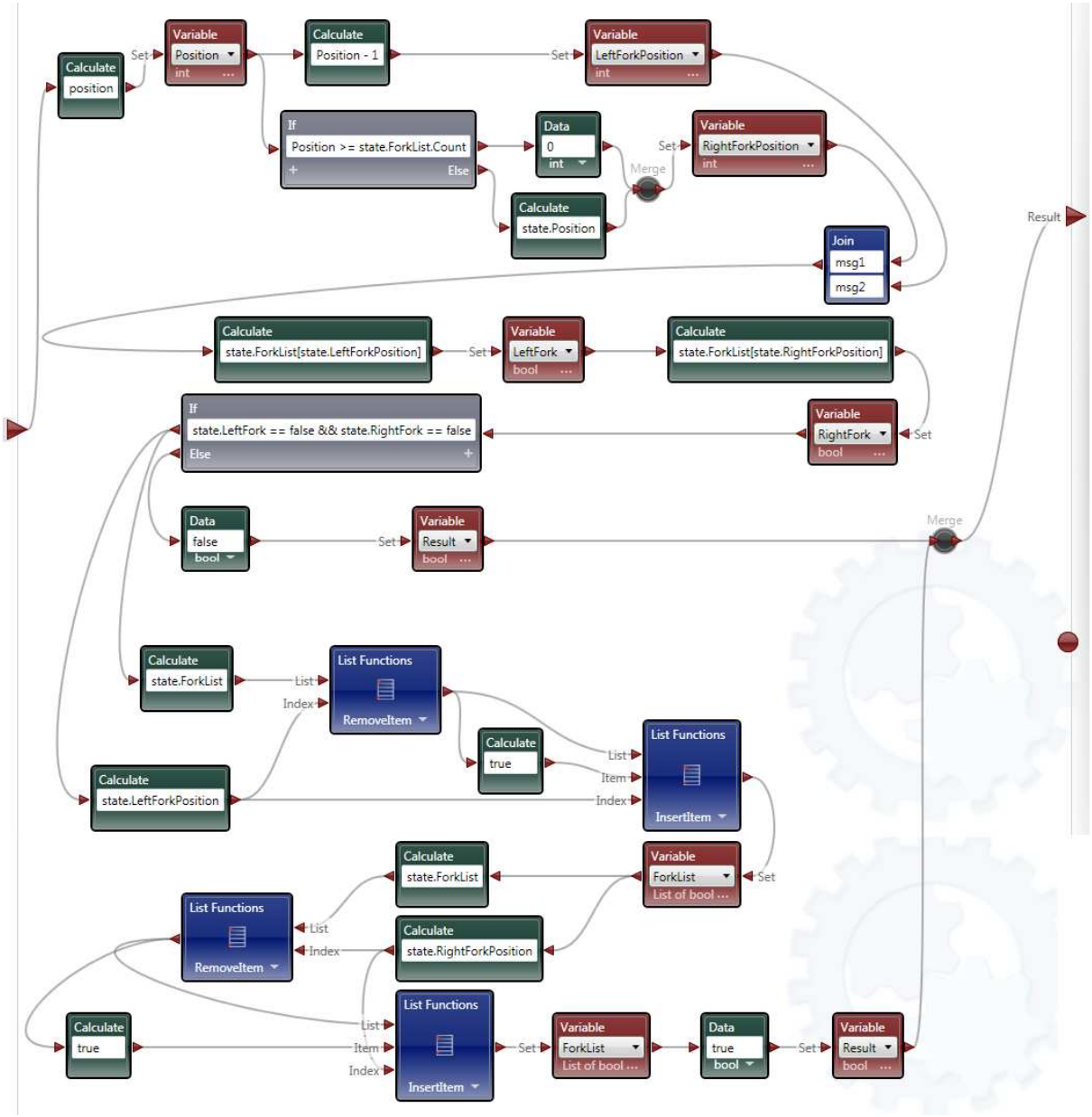



Obrázek 22: Akce GetPosition

#### 6.3.3.3 Akce GetForks

Akce GetForks, uvedená na obrázku 23, se bude volat pro zajištění vidliček. Proto se jako vstupní parametr zadá pořadové číslo filozofa sedícího u stolu. Na základě této pozice se naplní proměnná LeftFork a RightFork. Tyto proměnné budou obsahovat hodnoty vybraných prvků ze seznamu ForkList. Filozof s pořadovým číslem 1, bude mít jako levou vidličku prvek s pořadovým číslem 0 a jeho pravou vidličkou bude prvek s pořadovým číslem 1. Filozofové jsou tedy číslovány od jedné, ale vidličky jsou realizovány jako prvky seznamu s číslováním od nuly.

Dále dojde k otestování stavu levé a pravé vidličky. Pokud nejsou obě volné, tak výstupem akce bude hodnota false. Jsou-li obě volné, tak dojde k jejich odstranění ze seznamu ForkList, a přidání prvků s hodnotou true na stejné pozice, z jakých byly předchozí prvky odstraněny. Důvodem této operace je omezený výčet akcí, které můžeme se seznamem provádět. V případě, že obě vidličky byly volné, bude výstupem akce hodnota true.



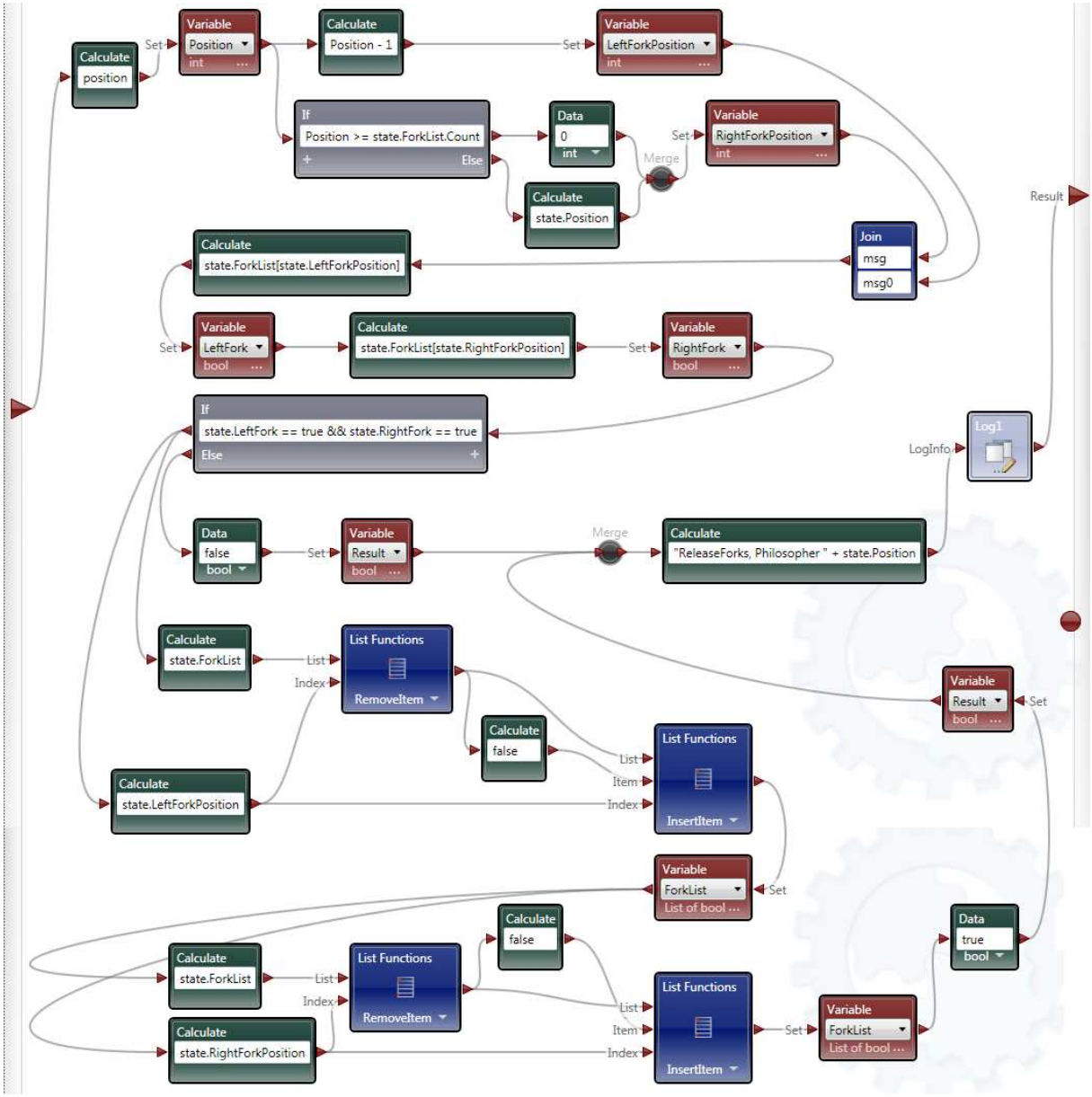

Obrázek 23: Akce GetForks

#### 6.3.3.4 Akce ReleaseForks

Pokud je potřeba přidělené vidličky uvolnit, tak se zavolá právě tato akce. Jako vstupní parametr se předá pozice filozofa u stolu. Z této pozice se zjistí index jeho levé a pravé vidličky v seznamu ForkList, a naplní se proměnné LeftFork a RightFork.

Pak se otestuje, zda vidličky které chceme uvolnit, jsou opravdu obsazené. Pokud by tomu tak nebylo, tak by výstupem akce byla hodnota false. Jsou-li obě obsazené, tak se vyjmou příslušné prvky ze seznamu ForkList, a na jejich pozice se vloží prvky s hodnotou false. Důvod, proč se prvky nejprve vyjmou, a potom na stejnou pozici vloží nové je podobně, jako v akci GetForks ten, že ve VPL máme omezený výčet operací, které můžeme se seznamem provádět. Po uvolnění vidliček bude výstupem hodnota true.

Abychom měli přehled o tom, který filozof zrovna požádal o uvolnění vidliček, tak se před ukončením akce, ještě vypíše informační hláška do logu. Vše ilustruje obrázek 24.

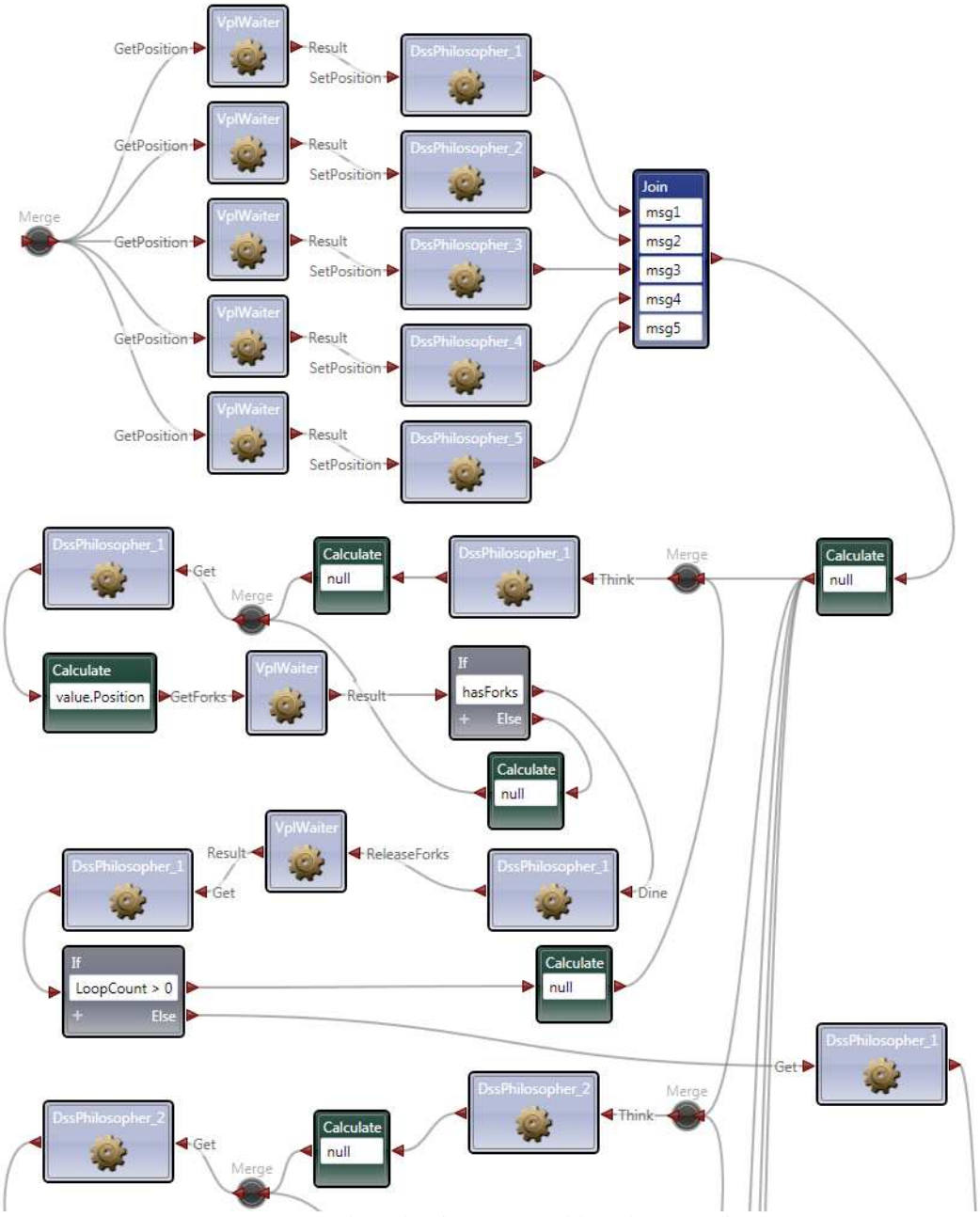



Obrázek 24: Akce ReleaseForks

### 6.3.4 Hlavní VPL diagram

Pomocí VPL diagramu se určí celková logika programu. Při jeho tvorbě můžeme využít nejen základních aktivit a při instalaci dodávaných DSS služeb, ale samozřejmě i námi vytvořenou aktivitu VplWaiter a službu DssPhilosopher. Jelikož námi řešený problém je kvůli počtu pěti filozofů rozsáhlý, tak si tento diagram rozdělíme na několik částí, které popíšeme zvlášť.

Celý program začne tím, že se pro všech pět filozofů, vyžádá od aktivity VplWaiter, pomocí akce GetPosition, přiřazení pořadového čísla filozofa u stolu. Toto pořadové číslo se pak zprávou SetPosition předá každému filozofovi, aby si jej mohl uložit. Aktivita Join nám zaručí, že program bude pokračovat až ve chvíli, kdy bude pořadové číslo přiřazeno všem filozofům. To je k nahlédnutí na obrázku 25.




Obrázek 25: První část hlavního VPL diagramu

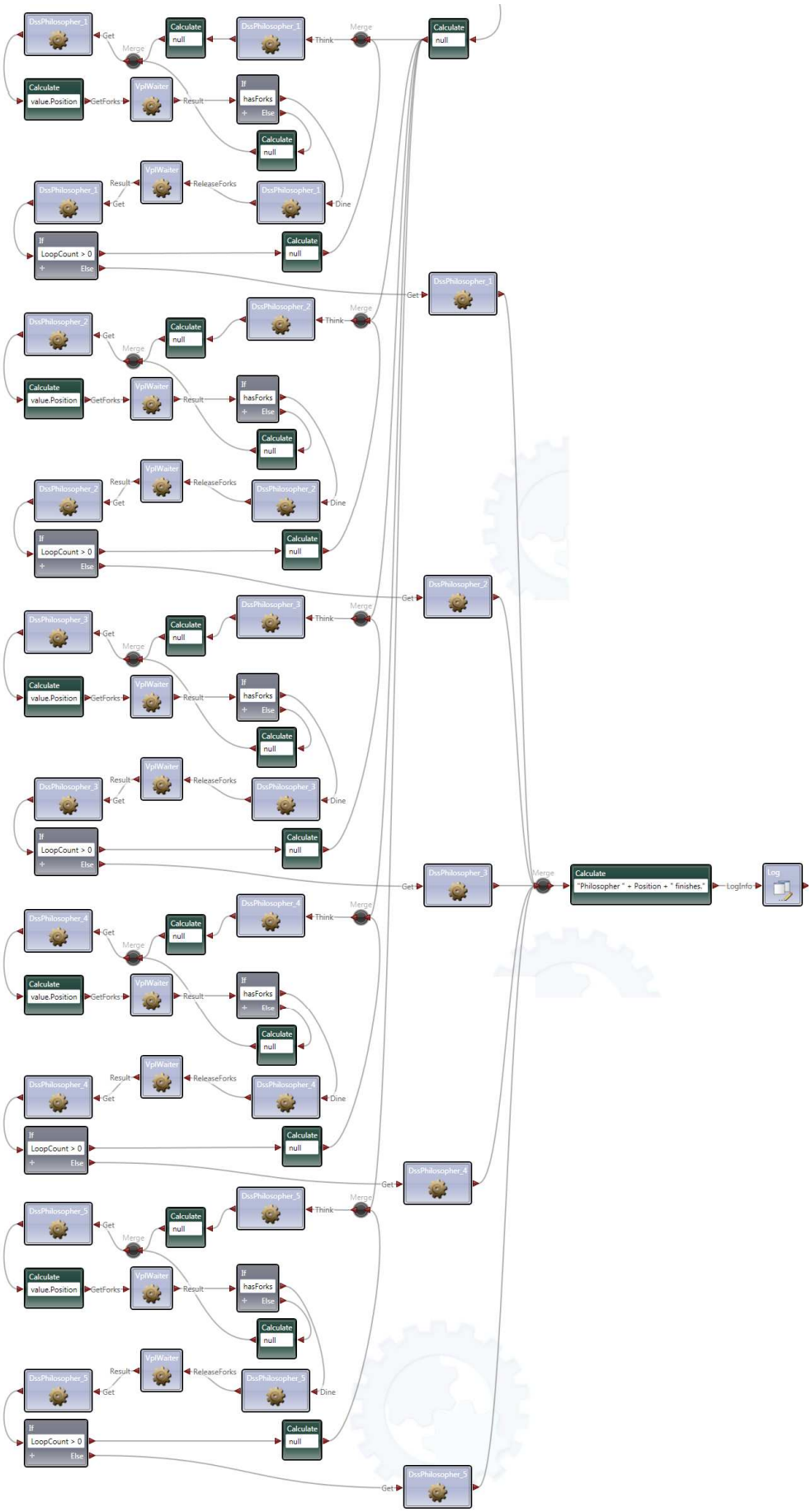
Potom se program rozdělí do pěti paralelních větví, kde v každé větvi probíhá cyklus přemýšlení a jedení. Na začátku cyklu se tedy odešle službě DssPhilosopher zpráva žádající o simulaci přemýšlení. Po uplynutí tohoto časového intervalu, se vyžádá informace o pozici filozofa u stolu, a ta se předá akci GetForks aktivity VplWaiter. Pokud se přiřazení vidliček nezdaří, neboť nejsou obě volné, tak se celá žádost o přiřazení opakuje.

Po úspěšném získání vidliček, je služba DssPhilosopher požádána zprávou Dine, aby simulovala jedení. Jak již víme, tak po uplynutí tohoto časového intervalu, dojde ve službě ke snížení hodnoty LoopCount o jedničku. Následuje tedy volání akce ReleaseForks aktivity VplWaiter, aby byly vidličky uvolněny, a mohl je případně použít některý ze sousedních filozofů.

Následné volání zprávy Get nám zjistí zbývající počet cyklů, které ještě zbývá vykonat. Má-li se celý cyklus znovu opakovat, tak se pokračuje opět zasláním zprávy Think.

Aktivity Calculate s hodnotou null jsou použity pouze z toho důvodu, že všechny vstupy do aktivity Merge musejí mít stejný datový typ. Pokud to nejsme schopni dodržet, tak lze toto omezení obejít právě například zasláním hodnoty null.

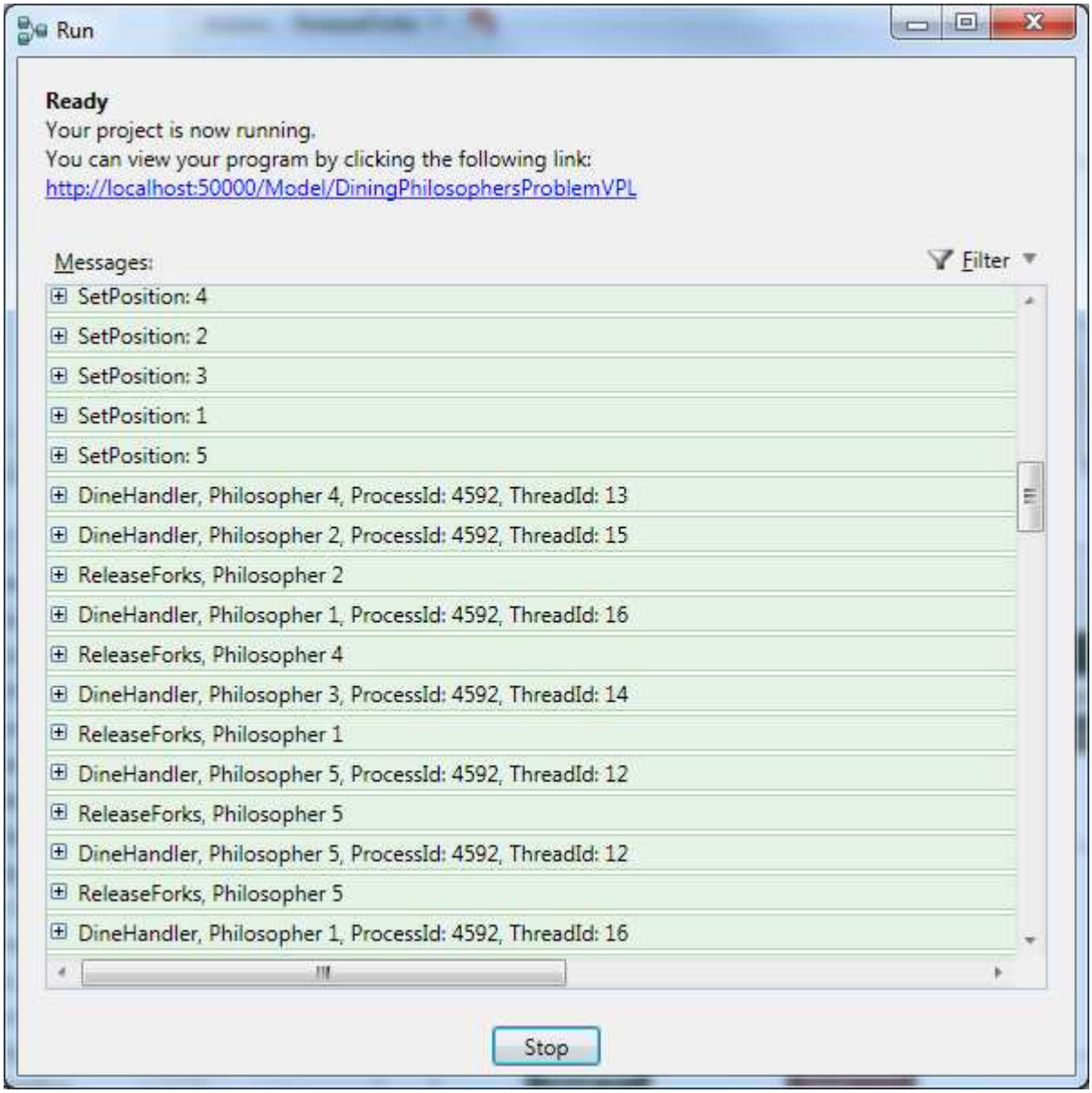
Druhá část diagramu, která je zobrazena na následující stránce, již jen zobrazuje podobný scénář, který se odehrává i pro ostatní filozofy. Co je však na něm navíc, je zobrazení toku po ukončení požadovaného počtu cyklů přemýšlení a jedení. V takovém případě se totiž zašle službě DssPhilosopher zpráva Get, která opět vrátí informaci o pozici filozofa u stolu, a ta se pak použije k zobrazení informace o ukončení činnosti filozofa. To je vidět na obrázku 26.




Obrázek 26: Druhá část hlavního VPL diagramu

### 6.3.5 Výstup programu

Po spuštění programu se nám zobrazí konsole obsahující základní informace o jeho běhu. Pro lepší představu o činnosti programu, se v metodě DineHandler() třídy DssPhilosopherService, loguje i informace o procesu a vláknu (viz obrázek 27).



Obrázek 27: Výstup programu

Na adrese http://localhost:50000/console/output, máme k dispozici ještě podrobnější informace, které se logují. Část je vidět na obrázku 28. Velmi důležitou informací je to, že se jednotlivé akce aktivity VplWaiter provádějí s atributem Exclusive. Pouze takto máme totiž zaručeno, že se o manipulaci s vidličkami nemůže pokoušet žádný jiný filozof dříve, než se vyřídí požadavek toho předchozího.

| | | | | |
|---|---|---|---|---|
| | 218 | 21:07:40 | * | Entering: program.activity0.GetForks: Exclusive |
| | 219 | 21:07:40 | * | Philosopher 5 finishes. |
| | 220 | 21:07:40 | * | Closed: program.activity0.GetForks |
| | 221 | 21:07:40 | * | Leaving: program.activity0.GetForks |
| | 222 | 21:07:40 | * | DineHandler, Philosopher 4, ProcessId: 4592, ThreadId: 13 |
| | 223 | 21:07:40 | * | Entering: program.activity0.GetForks: Exclusive |
| | 224 | 21:07:40 | * | Closed: program.activity0.GetForks |
| | 225 | 21:07:40 | * | Leaving: program.activity0.GetForks |
| | 226 | 21:07:40 | * | DineHandler, Philosopher 1, ProcessId: 4592, ThreadId: 16 |
| | 227 | 21:07:41 | * | Entering: program.activity0.ReleaseForks: Exclusive |
| | 228 | 21:07:41 | * | ReleaseForks, Philosopher 1 |
| | 229 | 21:07:41 | * | Closed: program.activity0.ReleaseForks |
| | 230 | 21:07:41 | * | Leaving: program.activity0.ReleaseForks |
| | 231 | 21:07:41 | * | Philosopher 1 finishes. |
| | 232 | 21:07:41 | * | Entering: program.activity0.ReleaseForks: Exclusive |
| | 233 | 21:07:41 | * | ReleaseForks, Philosopher 4 |
| | 234 | 21:07:41 | * | Closed: program.activity0.ReleaseForks |
| | 235 | 21:07:41 | * | Leaving: program.activity0.ReleaseForks |
| | 236 | 21:07:41 | * | Entering: program.activity0.GetForks: Exclusive |
| | 237 | 21:07:41 | * | Closed: program.activity0.GetForks |
| | 238 | 21:07:41 | * | Leaving: program.activity0.GetForks |
| | 239 | 21:07:41 | * | DineHandler, Philosopher 3, ProcessId: 4592, ThreadId: 14 |
| | 240 | 21:07:41 | * | Entering: program.activity0.GetForks: Exclusive |
| | 241 | 21:07:41 | * | Closed: program.activity0.GetForks |
| | 242 | 21:07:41 | * | Leaving: program.activity0.GetForks |
| | 243 | 21:07:41 | * | Entering: program.activity0.GetForks: Exclusive |
| | 244 | 21:07:41 | * | Closed: program.activity0.GetForks |
| | 245 | 21:07:41 | * | Leaving: program.activity0.GetForks |
| | 246 | 21:07:41 | * | Entering: program.activity0.GetForks: Exclusive |
| | 247 | 21:07:41 | * | Closed: program.activity0.GetForks |
| | 248 | 21:07:41 | * | Leaving: program.activity0.GetForks |
| | 249 | 21:07:41 | * | Entering: program.activity0.GetForks: Exclusive |
| | 250 | 21:07:41 | * | Closed: program.activity0.GetForks |
| | 251 | 21:07:41 | * | Leaving: program.activity0.GetForks |
| | 252 | 21:07:41 | * | Entering: program.activity0.GetForks: Exclusive |
| | 253 | 21:07:41 | * | Closed: program.activity0.GetForks |
| | 254 | 21:07:41 | * | Leaving: program.activity0.GetForks |
| | 255 | 21:07:41 | * | Entering: program.activity0.GetForks: Exclusive |

Obrázek 28: http://localhost:50000/console/output

### 6.3.6 Závěr

Právě se nám podařil vyřešit problém večeřících filozofů, pomocí jazyka Visual Programming Language, a to i s využitím služby napsané pomocí Decentralized Software Services. Při jeho řešení jsme se však museli vypořádat s omezeným výčtem funkcí, které lze vykonávat s proměnnými typu List, či s nutností mít všechny vstupy do aktivity Merge stejného typu. Také řešení, kdy se při neúspěšném vyžádání vidliček celý proces žádosti opakuje, nemusí být při řešení některých problémů, považován za zrovna ideální. Na druhou stranu jsme celkem jednoduše mohli využít faktu, že se v aktivitách jednotlivé jejich akce, vykonávají s atributem Exclusive. To nám zajišťuje, že se stav aktivity po dobu vykonávání akce, nezmění voláním nějaké akce jiné. Počet a sdílení vláken, lze nastavit pomocí atributu ActivationSettings, který jsme v našem příkladu použili u třídy DssPhilosopherService.

# 7 Instalace

Microsoft Robotics Developer Studio 2008 R3 lze volně stáhnout ze stránek firmy Microsoft [1]. Podporovanými operačními systémy jsou Windows 7, Windows Vista, Windows XP. Pro vývoj vlastních služeb pro MRDS, je zapotřebí použít Microsoft Visual Studio 2008 nebo 2010, které je nutné nainstalovat před samotným MRDS. Po ukončení instalace, je potřeba provést kompilaci nainstalovaných příkladů. To se provede spuštěním cmd souboru, který je dostupný v menu Start / Microsoft Robotics Developer Studio 2008 R3 / Build All Samples.

# 8 Závěr

Microsoft Robotics Developer Studio je primárně určeno na manipulaci s roboty, či simulaci jejich pohybu. Proto musí nabízet způsob, jak synchronizovat jednotlivé povely. To řeší tím, že nabízí Concurrency and Coordination Runtime, nad ním vybudované Decentralized Software Services a jako vizuální prostředek pro vyjádření toku dat, lze použít Visual Programming Language. To byl důvod, proč jsme se museli nejprve seznámit s CCR, DSS, a teprve potom s VPL.

Při řešení problému, který je v oboru informačních technologií znám pod pojmem Problém večeřících filozofů, bylo možné si vyzkoušet, jak by se tento problém mohl pomocí MRDS řešit. Je samozřejmé, že zde uvedené řešení není zdaleka jediné, které lze použít. Stačilo to však na to, aby bylo možné konstatovat, že přestože MRDS představuje celkem komplexní řešení, tak jsou situace, kdy je potřeba se nad vhodností jeho použití přinejmenším dobře zamyslet. Rozhodně nelze říci, že by bylo možné pomocí MRDS nahradit dobré znalosti z oblasti procesů, vláken a jejich vzájemné synchronizace.

Jazyk Visual Programming Language se ukazuje jako mocný nástroj, především při sestavování několika různých DSS komponent do větších celků. Avšak rozhodně jej nelze považovat za rovnocenný prostředek k budování rozsáhlých systémů, kterým bychom nahradili klasické programování. Lze si však představit situaci, kdy se s jeho pomocí sestaví nějaké prototypové řešení, které se pak překompiluje do DSS, a vše se pak dokončí klasickým programováním.

Našeho cíle, seznámit se s Microsoft Robotics Developer Studiem a pomocí Visual Programming Language demonstrovat jeho možnosti, bylo tedy dosaženo. Nutno však podotknout, že tím se možnosti MRDS ani zdaleka nevyčerpaly. Například jsme se blíže neseznámili s Visual Simulation Environment, sloužícím k vizuální simulaci prostředí, a k pohybu robotů v něm. A právě to je jedna z oblastí, kterou by se mohlo na naše snažení, o pochopení možností MRDS, navázat.

# 9 Literatura

[1] *Microsoft Robotics Developer Studio 2008 R3*
URL: <http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=c185a802-5bbe-4f28-b448-aefe63a7eff7> [cit. 2010-01-07].

[2] *Microsoft Robotics Developer Studio Online Documentation*
URL: <http://msdn.microsoft.com/en-us/library/bb881626.aspx> [cit. 2010-04-19].

[3] *Concurrency and Coordination Runtime*
URL: <http://msdn.microsoft.com/en-us/library/bb905470.aspx> [cit. 2011-01-22].

[4] *Decentralized Software Services*
URL: <http://msdn.microsoft.com/en-us/library/bb905471.aspx> [cit. 2011-02-04].

[5] *Visual Programming Language*
URL: <http://msdn.microsoft.com/en-us/library/bb964572.aspx> [cit. 2011-02-15].

[6] *Procesy / 2.3.1 Večeřící filozofové*
URL: <http://poli.cs.vsb.cz/edu/osy/auth/procesy.pdf> [cit. 2011-02-17].

[7] *Threads in C#: Dining philosophers*
URL: <http://www.complore.com/threads-c-dining-philosophers> [cit. 2011-02-23].

# Příloha A

Přiložené CD obsahuje složky a soubory, uvedené ve výpisu 54.

```
DiningPhilosophersProblemSolution
        ConcurrencyAndCoordinationRuntime
                Properties
                        AssemblyInfo.cs
                app.config
                ConcurrencyAndCoordinationRuntime.csproj
                Philosopher.cs
                Program.cs
                TableWithInterleave.cs
        DssPhilosopher
                Properties
                        AssemblyInfo.cs
                Proxy
                        DssPhilosopher.Y2011.M04.Proxy.cs
                        DssPhilosopher.Y2011.M04.Transform.cs
                        Microsoft.Resources.DssModel.dss
                DssPhilosopher.cs
                DssPhilosopher.csproj
                DssPhilosopher.csproj.user
                DssPhilosopher.manifest.xml
                DssPhilosopherTypes.cs
        ThreadsInCSharp
                Properties
                        AssemblyInfo.cs
                app.config
                Forks.cs
                Philosopher.cs
                Program.cs
                ThreadsInCSharp.csproj
        ConcurrencyAndCoordinationRuntime.psess
        DiningPhilosophersProblemSolution.sln
        DiningPhilosophersProblemSolution.suo
        ThreadsInCSharp.psess
DiningPhilosophersProblemVPL
        Diagrams.xml.compressed
        DiningPhilosophersProblemVPL.mvpl
        DiningPhilosophersProblemVPL.mvpl.user
DP_Nov718.pdf
```

Výpis 54: Obsah CD